AVA User's Manual
`version 0.2.4`

Uroš Platiše

12. March 1999

Contents

Chapter 1

Introduction

Even though small micro-controllers are getting stronger and stronger every day there is still a lot of code to be written in pure assembler. For example kernel itself, device drivers, fast interrupt routines, optimized mathematics functions, etc.

AVA (assembler and linker) was mainly designed for 8 and 16 bit micro-controllers but can be easily extended to 32 bit families also. In addition to standard assemblers for micro-controllers it features powerful segments and *virtual* symbols. These two features improve modular programming and allow objects to be completely independed from each other.

AVA integrates two packages:

1. Assembler and
2. Linker

Depending on the command line parameters it executes either the first or the second one. Libraries are supported by the ordinary objects where each function can be declared in its own segment which is removed if not needed.

AVA User's Manual is divided into the following chapters:

1. Installation
   Gives references to downloading locations, compiling and installation procedure.

2. Running the AVA
   This chapter describes input/output streams and command line options.

3. Assembler Language
   Explanation of Preprocessor, Symbols, Segments and other Keywords.

4. Programming in AVA
   This chapter is more of practical interest. It is important that AVA is used in a proper way because only in that case the user can get maximum from it.

Chapter 2

Installation

AVA is distributed as a source code and should compile on most of the UN*X systems. Hardware and software requirements for compilation and installation procedure are:

- GNU C++ compiler version 2.7.2.1 or higher.
- 3 MB of hard disk space
- 8 MB of memory with swap is suggested
  It also compiles on 4 MB systems in about one hour and more

Till now AVA was tested and used on Debian and Redhat Distributions.

2.1 Obtaining the AVA

AVA can be downloaded from primary web site:

```
http://medo.fov.uni-mb.si/mapp
```

Mirror sites are not available yet.

If you do not have former versions of AVA you have to download complete .tar.gz package and not just *diff* files. After you download the latest release extract it as:

```
tar -zxf ava-x.y.z.tar.gz
```

Enter the `ava-x.y.z` directory and you shall see the following directory structure, where:

`.`	info files
`avalib`	AVA library
`src`	assembler source code
`doc`	document source
`examples`	assembler examples

2.2 Configuring

By the default AVA library is copied to the

```
/usr/local/uTools/ava
```

directory and `ava` executable to the

```
/usr/local/bin
```

directory. If you disagree with the defaults, please edit the:

```
ava-x.y.z/src/Makefile
```

and change directories to your requirements. This settings affects both, compilation and installation procedure.

Postscript and HTML version of the document is copied to the

```
/usr/doc/uTools/ava
```

You can change the default settings in the:

```
ava-x.y.z/doc/Makefile
```

2.3 Compiling

Since AVA is a console application there are actually no parameters to be set by users individually. Simply enter the directory `ava-x.y.z/src` and type:

```
make
```

Note AVA should compile without errors and warnings. If any warning is present regarding to the AVA source please report bug as described in Bug Report section B.2

2.4 Installation

To install software change directory to `ava-x.y.z/src` and type:

```
make install
```

After installation you may enter the `ava-x.y.z/examples` directory and type:

```
ava endian.s
```

to compile and:

```
ava endian.o
```

to link it. If this example did not work, compilation or installation procedure has failed. By the default the executable code is put into `a.out` file unless specified otherwise.

To install documentation in Postscript and HTML formats, change directory to `ava-x.y.z/doc` and type:

```
make install
```

Chapter 3

Running the AVA

AVA is a console application running in batch mode. It reads input either from a file or from the standard input (stdin). Output is written to another file or to the Standard Output (stdout). By default error messages are printed to the Standard Error (stderr).

3.1 I/O Streams

Assembler and linker are integrated. Which of them is invoked is identified from the source filename extension. If Standard Input (stdin) and/or Standard Output (stdout) are specified AVA automatically invokes Assembler. Linker cannot use these two streams as input/output streams.

The following suffixes of source file names indicate kind of processing to be done:

suffix	**input/output, type, action**
`.s`	i, assembler source, assemble
`.o`	i/o, object, link
`.inc, .h`	input, include
	not usually named on command line
`.lst`	o, listing report
`a.out`	o, linker default output

Assembler can assemble one file at a time only.

3.2 Command Line Options

General syntax of the AVA command line parameters:

```
ava -option1 ...  -optionn   file1.x ...  filen.x
```

One option can be specified after '-' character only. Full path name should be named for all files that are not found in the current directory.

Note The order of options does not matter but the order of files is very important.

Files are linked in exactly the same sequence as listed in the command line. This has strong effect on virtual symbols. Please refer to section 4.3.3 for details.

Several options are available to:

- get on-line help,
- set the assembler and linker behaviours and
- to control reports.

3.2.1 Help

AVA options described in the following two sections are also documented on-line, using the -h switch. For example type:

```
ava -h
```

3.2.2 Assembler and Linker Options

option	**description**
-o *outfile*	Redirects output to the *outfile*. Hint: if *outfile* equals *stdout*, standard output is used. *example:* ava -o project main.o print.o
-p	Use stdin as input and stdout as output unless -o is specified. *example:* ava -p <file.s | gzip >file.o.gz
-A*device*}	Defines target device *device*.
-D*macro*{=*val*}	Defines macro with value *val*. If *val* is not specified 1 is assumed. *example1:* ava -DVER=2 main.s *example2:* ava -DSTR="\"string\"" main.s
-f*macro*{=*val*}	Defines public macro with value *val*. To macro two underlines are inserted at the begining. If *val* is not specified 1 is assumed. *example1:* ava -favr_noskipbug main.s
-I*directory*	Add include directory to the search list. *example:* ava -I../include main.s
-I*filename*	Include *filename* to the source. *example:* ava -I../include main.s
-T{*filename*}	Include *filename* to the source. if *filename* is not given *target.inc* is used. Thid file is the first file in the queue - before those specified by the -I*filename* option. It is searched in the default directory and then on the predefined list. See section 4.2 for details.

3.2.3 Reports Options

option	**description**
`-v`	Turn on info messages. (Assembler and Linker status, Segment Tree, ...) *example:* `ava -v main.s`
`-L`	Turn on listing reports for file being assembled. *example:* `ava -T main.s`
`-l`*logfile*	Redirect segment tree information and status reports to the file. Errors and warnings are printed to both, stderr and *logfile*. *example:* `ava -lava.log main.s`

Options that take special effect on assembler source are further explained in the next chapter.

Chapter 4

Assembler Language

4.1 Input

Assembler source consists of elements called tokens. The token may be one of the following: numbers, string, quoted string, preprocessor directives, math and control symbols and remarks. To each type of token a range of valid characters is declared. If token does not match that range lexer produces an error and assembler terminates.

The types of tokens and ranges are represented in the table below.

token	**input characters**
decimal number	{0..9}
hex number	$\$\{0..9,a..f,A..F\}$, 0x{0..9,a..f,A..F}
string	{a..z,A..Z,_}{a..z,A..Z,0..9,_},...
quoted string	"..."
preprocessor	♯*string*
math symbols	$\{+,-,*,/ =,<,>,\&,\vert,\wedge\}$
control symbols	{., , ,{,},(,),\ }
standard C remarks	/* ... */
standard asm remarks	; ...

4.1.1 Expressions

Expressions are made of numbers, strings, math and control symbols. AVA expression solver understands the following operations:

symbol	**operation**
$+, -$	add, subtract
$\&, \vert, \wedge, <<, >>$	and, or, xor, logical shift left, right
$*, /$	multiply, divide
$<, >, ==, <=, >=$	less, greater, equal, less/greater or equal

The general form of expression:

`((A` *`operation`* `B)` {*`operation`* `C`}`)`

where `A,B` and `C` may be expressions also.

4.2 Preprocessor

4.2.1 Directives

AVA provides C like preprocessor. Preprocessor directive starts with ♯ character followed by a command. The table below lists the supported directives:

directive	**meaning**
♯arch *target_device*	Defines target device.
♯include "*filename*"	Includes *filename* to the current source.
♯adddir "*filename*"	Adds alternate include directory to the list.
♯define ...	Defines macro.
♯undefine *macro_nm*	Undefines macro *macro_nm*.
♯ifdef *macro_name*	If macro *macro_name* is defined, enter.
♯if *expression*	If *expression* is true, enter.
♯else	`♯ifdef` *macro_name* `... ♯else ... ♯endif.`
♯error *message*	Generates error and report message.
♯print *message*	Prints info message.

Directive ♯arch *target_device*

Defines target device. This directive defines macro *target_device* and includes the file *../uTools/ava/arch.inc*. File *arch.inc* declares memory model, actual device, memory mapped I/Os, ...etc. See content of the *arch.inc* for details.

Example 1

```
/* set device */
#arch AT90S1200
```

Target device can be also declared by the -A switch.

Directive ♯include "*filename*"

Includes *filename* to the current source. The file specified is first searched in the current directory and if it is not found, AVA continues searching in the alternative directories as specified by the -I*directory* option on the command line or by the ♯adddir "*filename*" directive in the source code.

Example 2

```
/* define model */
#arch AT90S1200

/*
   Include standard bit definitions according to the
   selected device.
*/
#include "avr.inc"
```

Directive ♯adddir "*filename*"

Adds alternate include directory to the search list used by the ♯include directive. This directive is mostly used by the Target Definition File.

Example 3
```
/* target.inc */

/* Set additional include directories ... */
#adddir "../csys/include/asm"
#adddir "../common/include/"
```

Directive ♯define *macro_nm*

Defines macro with name *macro_nm*. Since macros are a little bit extended they are explained in the separate section 4.3.3. See example 4 and 5.

Directive ♯undefine *macro_nm*

Undefines macro *macro_nm*. See example 4 and 5.

Directive ♯ifdef *macro_name*

Enters the ♯ifdef ... ♯endif block if macro *macro_name* is defined.

Example 4
```
#ifdef __NICE_LEVEL
#  define __NICE_LEVEL 49
#endif
```

Directive ♯if *expression*

Similar to ♯ifdef except that expression may be given. All undefined macros within the expression are treated as zeros.

Example 5
```
#if (__NICE_LEVEL < 41)| (__NICE_LEVEL > 58)
#  undefine __NICE_LEVEL
#  define __NICE_LEVEL 49
#endif
```

Directive ♯else

Else can be used in conjunction with ♯ifdef directive. General form is: `♯ifdef` *`macro_name`* `... ♯else ... ♯endif`.

Example 6
```
#ifdef __NICE_LEVEL
#  define __NICE_LEVEL 49
#else
```

```
#  undefine __NICE_LEVEL
#  define __NICE_LEVEL 49
#endif
```

Directive ♯error *message*

Generates error and report message.

Example 7

```
/*
   Use of parethness is recommended because of the macro
   replacements.
*/
#ifndef __NICE_LEVEL
#  error "__NICE_LEVEL is not defined!"
#endif
```

Directive ♯print *message*

Prints info message. Message is terminated by a new a line.

Example 8

```
/* Prints __NICE_LEVEL to the stdout and log file */
#print "__NICE_LEVEL = " __NICE_LEVEL
```

4.2.2 Architecture Definition File

The Architecture Definition File (*arch.inc*) is a part of the AVA library. This file includes all the information of the supported models and families.

The file *arch.inc* is automatically included after the ♯arch preprocessor directive.

Note Never include or modify *arch.inc* file unless all consequences are known.

4.2.3 Target Definition File

AVA is a segment oriented assembler. Each piece of code may be placed in any kind of segment. Such a mechanism provides maximum control over the code and prevents from undesired errors (see section 4.4 for detailed explanation on segments).

One of the mechanism that helps to keep things in order is the Target Definition File or *target.inc*. For larger micro-controller systems with memory mapped I/Os, "huge" software, the *target.inc* file should provide complete hardware description. Using the -T option in the command the *target.inc* file will be included to each source automatically.

Of course, adding the line `♯include "target.inc"` to each source has the same effect but -T option might be used as a shortcut.

One of the most important things that the linker should be instructed is about memory holes. If linker is not notified about them, it cannot avoid these holes and data segment **may be** improperly placed! Please see examples in the chapter 5 and section 4.4 for help on declaring the memory holes.

4.3 Symbols

There are two kind of symbols:

- Macros and
- Labels

To label offset is assigned and macro can be used to replace expressions and/or assembler code.

4.3.1 Attributes

The present section describe general properties of the macros and labels. Attributes declare the different behaviours of the symbols during compilation and linkage process. The attributes are:

attribute	**meaning**
internal	Declares internal symbol. This symbol is declared within current object file only and is completely invisible from the outside. *This is the assembler default attribute.*
public	Create public symbol. Public symbol is visible inside and outside of the current object.
extern	Creates reference to other objects. Public symbols defined in other objects are visible to current object if and only if references exist.
virtual	Creates virtual symbol. This special flag allows further redefinition of the symbol. The value and attribute of the symbol can be redefined by another definition of the same symbol.

Symbol can have assigned multiple properties. But not all of them are valid. For example; symbol declared as (public, virtual) is invalid but (extern, virtual) is right. Note that attribute (extern) creates references outside the current object and (public) only exports the symbol. Virtual symbol can be only replaced by outside symbol if reference exist.

The following combinations make sense:

attribute	**meaning**
none	Creates internal symbol.
public	Creates public symbol.
extern	Creates reference to other objects.
virtual	Creates internal virtual symbol. *This declaration makes sense for development purposes only.*
virtual extern	Creates global virtual symbol.

4.3.2 Label

To label offset is assigned. General form of declaration:

```
attributes label_name{:}
```

The semicolon ':' is required only if there is no attribute assigned to the label.

4.3.3 Macro

Macro replaces numbers with constants, expressions or even larger parts of code. General form of declaration:

```
♯define attributes macro_name(a0,a1,...,an) macro_string
```

If *macro_string* does not fit into one line, use the \ character on the end of each non-last line. Also the parameters $a_0, a_1, ...$ and on may be split into two or more lines. Additional character on the end of the line is not needed since every parameter is terminated either by coma or) character.

The general form can be written as:

```
♯define attributes macro_name(a0,a1,...,an) macro_string0 \
macro_string1   \
...
macro_stringn-1 \
macro_stringn
```

Note ava-0.1.8 does not yet support sections and labels. They should not be used within the macro.

4.4 Segments

The importance of the full control over the code is crucical and cannot be explained in one hour course. Specially if the target device has limited memory - okay, every machine on the planet is limited, but 1 or 8 kb of memory is something heavily under the normal range people know it from the PCs.

Treatment of segments is heavily extended in comparison to the other assemblers and linkers. The standard command

```
org offset
```

does not exist. Better result is achieved by using the higher order segments.

The AVA segments form multi-children tree structure just as like a file system. The *root* of the segments represents target its children define physical memory areas. Moreover this areas can be even expanded to abstract areas but let us describe them later on.

Simple representation of the program for the ***AVR*** micro-controller is shown in figure 4.1. Here the *flash, eeprom* and *eram* are ***AVR*** base segments. They declare physical memory models and the downloading tool is expecting this segments.

From some reasons it makes sense to define the segment order:

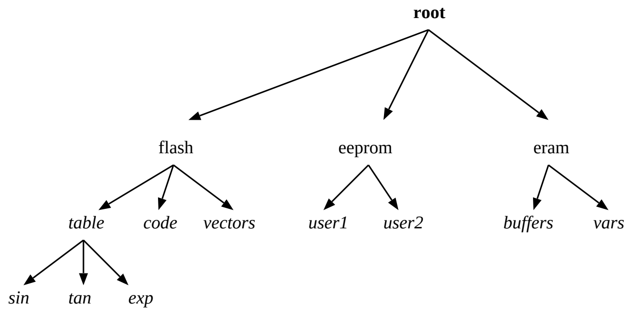

Figure 4.1: An example of a segment structure.

> *The segment order S_O of the segment S is the number of vertices on the shortest path from the segment S to the* **root**. *The vertex of the segment S is also included.*

Now we can say: base segments are segments of the first order.

4.4.1 Attributes

The following segments attributes are available. Each attribute is described in greater details in further sections.

attribute	**meaning**
abstract	Declare abstract segment that never appears in the output.
removable	Allow linker to remove that segment if it is not used.
abs=*val*	Absolute origin of the segment.
size=*val*	Fixed size of the segment.
mirror=*segment*	Declares mirror segment.
align=*val*	Keep segment aligned to *val* bytes.

Multiple attributes can be assigned to single segment.

Abstract Segment

If we take a look at our previous example in figure 4.1 we see segment named `eram`. In any case, these segment has nothing to do in the output for the ***AVR*** micro-controller since downloading tool cannot load data to the volatile memory.

Every segment that is used for internal processing only is called *abstract* segment. This segment exist as such until the linker is finished.

The mechanism of abstract segment can be generalised and can be used for any kind of processing. For example: magic tables. The user can specify new segment to which every device driver defines a byte with a corresponding label. After linkage process to every byte an unique address is assigned and thus unique magic number in the corresponding label.

Removable Segment

The *removable* attribute efficiently extends object files to libraries. Each function should be given be placed in its own higher order removable segment which consist at least **ONE** public label and **not** macro. If this public label was used at least once by other source the segment will be linked together with the rest of the code. Otherwise it is removed.

Absolute Origin

This attribute sets the final position of the segment and it must be specified to all first order segments. The segments which are not absolutelly positioned are called floating segments and are placed by AVA linker. The complete segmentation report is printed to the log file or to the standard output if log file is not given.

Fixed Size

If the size of segment is not given the length may vary according to the usage. These segments are called variable segments. Because the root segment does not know the lengths of the first order segments their maximum size must be defined.

Note Every higher order segment (children) becomes variable segment if its size is not specified.

Mirror Segment

Let us go back to the example again where we already said few words about the `eram` segment. Now imagine the second case

> We declare some data within the `eram` segment and would like to get it there, anyhow, after the ***AVR*** wakes up.

This typical example shows how does the C compiler work. The GNU C compiler stores data in second order segment named `gcc_data_section`. Linker copies this data to flash memory during the linkage process and checks for boundaries. The new segment in the flash memory is called *mirror* segment. The mirror segment must be declared before it is referenced.

Align Option

Sets alignment factor of the segment length. It can effect variable size segments only. The default segment alignment is set to 1 byte.

4.4.2 General Declaration Form

Before we introduce the general form of the segment declaration line the general form of the segment name must be defined. Going back to our example we define:

- each path is represented by a dot and
- vertex by its name

Since `root` segment and the first path are always present it makes sense we skip it. We get:

segname = S_1.S_2.S_3...

And the general form of the segment:

```
seg attributes segname
```

The same form initializes or sets the segment. See examples in chapter 5.

Example 9

```
/*
   Create and set the second order floating and variable
     segment to the eram segment named: data
*/
        seg eram.data

/*
   Map I/O memory ports to the eram segment of the AVR MCU.
*/
        seg abs=0 size=0x20 eram.registers
```

4.5 Keywords

The present section represents additional keywords for controlling the data storage and special keywords used by Architecture Definition File.

4.5.1 Define Storage

General declaration form:

ds.x n

Reserves space in a segment with length n times the size of the data type x.

Example 10

```
/* reserve 32 bytes of space in eram.data segment */
        seg     eram.data
buf:    ds.b    0x20

/* and 20 words for table */
tbl:    ds.w    20
```

4.5.2 Define Constant

General declaration form:

dc.x {number,string},{number,string},...

Initializes constants in a segment of data type x.

Example 11

```
/* New Line Chatacter */
#define NL      10

/* Hello Message (split in several lines) */
        seg     flash.data
hello:  dc.b    "Hello LED!",NL,
                "Hello again.",NL,
                0

/* store address of the label for indirect call */
/* Note: PC counts words and not bytes (LSB must be removed) */
putc_f: dc.W    _fputc >> 1
```

4.5.3 Data Types

AVA understands the following data types:

data type	meaning
b	byte
w	little endian word
W	big endian word
l	little endian long
L	big endian long

Little endian constants are stored in the form

$$B_{LSB}B_1...B_{n-1}B_{MSB}$$

and big endian as:

$$B_{MSB}B_{n-1}...B_1B_{LSB}$$

where MSB is the most significant byte and LSB the less significant byte.

4.5.4 Device

General declaration form:

```
device AVA _Arch_Modul parameters ...
```

This keyword is used by the Architecture Definition File only.

4.6 Instruction Set

Please refer to the micro-controllers data book for complete list and explanation.

Chapter 5

Programming in AVA

This chapter introduces simple examples and solutions some common problems. Examples are included in the `examples` directory.

5.1 Hello World!

The first typical example usually introduced by C and other languages is "Hello World!". Starting on a new micro-controller this "Hello World!" example is not that small piece of code. Driver for the UART/RS-232 module needs to be written, simple data structure as FIFO and printf function is required. All together counts about few hundreds bytes of code what is far too much for the first assembler program.

Let us better try to turn on and off a light emitting diode (LED) using the AT90S1200 ***AVR*** chip. Before going on please read chapter 4 which describes Assembler Language in details and section A.1 for ***AVR*** dedicated features.

5.1.1 Declarations

There is one declaration that must be written in every source before first line of code appears. This is the target selection. Example:

```
#arch AT90S1200
```

According to the ***AVR*** family (section A.1) we see that by default `flash` memory segment is selected. Thus every line after the ♯arch ... directive will appear in the flash memory.

It is more safe to put each type of code into a separate segment of higher order which name reflects the purpose of that segment. Thus program could be put in the segment `flash.code`.

And that is all.

5.1.2 Hello LED!

The following code turns on the led on port PB0 by setting the PB0 pin low.

```
/* examples/helloled.s */

/* set target device; see arch.inc for other models. */
#arch AT90S1200

/* Load include file of predefined port definitions.
   This file should not be included until device is selected.
   See inside of the avr.inc for details.
*/
#include "avr.inc"

/* general declarations */
#define LED1        PB0

/* calculate port B direction */
#define PORTB_DIR   BV(LED1)

/* register naming */
#define Tmp         r16

/* AVR AT90S1200 vectors start at address 0 */
    seg abs=0 flash.code

    rjmp __reset_   /* this vector is executed on every reset */
    reti
    reti
    reti

/* Initialize hardware ports */
__reset_:
    ldi Tmp, PORTB_DIR  /* set port B direction */
    out DDRB, Tmp

    clr Tmp             /* clear outputs and disable pull-ups */
    out PORTB, Tmp

loop:
    rjmp loop           /* loop for ever */
```

Even though this example turn on the LED only it includes all the properties of a larger software written in assembler. When you write in assembler take special care on:

- write remarks everywhere the code does not reflect the meaning visible from 100 meters.
- use macros to replace any kind of number (constant)
- use macros to keep your code independed of the hardware configuration - as LED1 in above example

- use segments more extensively - if there is table present in the flash memory do not put it in the `flash.code` segment but `flash.sin_table` for example.
- assign name to every register!

There are some rules which make assembler source even more clean.

5.2 Using the Target Declaration File

To the *target.inc* all hardware specifications should be given. The purpose of this file is to keep thousands lines of assembler code split in several files so far organised. Files that use *target.inc* can be compiled with -T option which automatically includes the *target.inc*. Otherwise the preprocessor line:

```
include "target.inc"
```

present in each source of the project does completely the same. The -T switch forces the Target Declaration File to be read first. Follow the files given by the -I switch and then the assembler source.

An example is given in `examples` directory and prototype of the *target.inc* file is located in `avalib` directory (or in `/usr/local/uTools/ava/target.inc`).

5.2.1 Target Declaration

The *arch.inc* file specifies the default parameters of the memory segments of the ***AVR*** family. If the target provides more memory the linker must be instructed. Otherwise it cannot be used.

Several macros are provided within the *arch.inc*. To change size of the RAM segment macro `__ERAM_SIZE` is provided as described in section A.1.

Even though AVA linker requires that only one object has set its segment size to the maximum value it is better to put that common properties in common file - the *target.inc*.

Example of target declaration:

```
/* Declare Size of the eram segment
  (before device declaration!) */
#define __ERAM_SIZE     0x1002

/* Declare MCU (AT90S8515 has enabled Wrap Around Option) */
#arch AT90S8515
```

5.2.2 Memory Mapped Ports

The next **very** important thing are memory mapped ports. AVA linker cannot know where the target device has memory mapped ports if you do not declare them.

Memory mapped ports are declared as higher order segments of fixed size and fixed absolute address. Example:

```
/* Memory mapped IO ports of my test board */
#define _IO_PORTS       0x1000

/* Declare segments. Protect eram segment to clash with
   the following io registers. */
seg abs=_IO_PORTS size=0x2 eram.ioports

/* Register Names: */
#define ADC_DATA        _IO_PORTS
#define ADC_CONTROL     _IO_PORTS+1
```

Besides segment declaration macros of the register names should also be provided.

Appendix A

Supported Families

The following specifications are provided as advanced information only and they are already considered in the `arch.inc` file.

A.1 ***AVR*** Family Micro-Controllers

AVA supports full range of ***AVR*** family micro-controllers. In general they can be all sorted in three groups.

Group	**Models**
1	AT90S1200
2	AT90S2313, AT90S2323, AT90S2333, AT90S2343 AT90S4414, AT90S4433, AT90S4434 AT90S8515 AT90S8535
3	ATmega603, ATmega103
4	*Not yet available.*

The ***AVR*** device is selected by writing the:

```
device AVR group_number
```

in the source code.

A.1.1 ***AVR*** Groups

Besides memory models ***AVR*** micro-controllers have different number of instructions. The following table represents the difference between first/second, second/third and third/fourth groups.

Group	**Instructions**
1/2	adiw, icall, ijmp, ld/st *X*, ld/st *Y*, ld/st -Z+, ldd/std, lds, lpm, pop, push
2/3	call, jmp
3/4	mul

A.1.2 Memory Models

The following segments are provided within the arch.inc:

Group	HW Segments	Abstract Segments
1	`flash`, `eeprom`	
2, 3	`flash`, `eeprom`	`eram`, `eram.registers`, `eram.io_space`

Segment sizes depend on a chosen model. Segments `eram.registers` and `eram.io_space` are of fixed size and have absolutely positioned segments.

The user should change the size of the RAM - `eram` segment. By the default, minimum values are used for all models - what includes internal SRAM only. Before declaring the device set `__ERAM_SIZE` as shown in the following example:

```
#define __ERAM_SIZE 0x8000
#arch AT90S8515
```

As suggested by the AVA this kind of declarations should move to the target specific *target.inc* file.

By default, segment `flash` is selected.

A.1.3 Special Flags

Note Flags must be declared as public macros.

The following flags are supported by the ***AVR*** family:

AVR_WRAP_AROUND Relative addressing capabilities of the relative calls can handle only up to 4 kb of memory in negative and positive directions. With this option, the top and the bottom addresses are joined together as they were linear and thus virtually extends the addressing space of the relative calls. This option was provided for 8 kb flash models only and allows `rjmp` instruction to cover the entire program memory.

avr_noskipbug Some AVR-CORES are found to have bug when pushing the program return address on interrupt occurence. This happens if and only if interrupt vector is executed while one of the skip instructions (sbrs,sbrc,sbis,sbic and cpse) is being executed in its first or second cycle and if the instruction followed by the this instruction is the two-word (32 bit) instruction such as lds,sts,jmp and call. Moreover, this bug occurs if and only if address of the two-word instruction points beyond 0x400.

The AVA assembler performs check and reports warning if such a bug might happen. Since later revisions of the AVR MCUs have fixed that bug you might need to append the option

```
-favr_noskipbug
```

to the command line in order to disable the AVA bug hunting mechanism.

Appendix B

Support

B.1 New Releases

New releases, updates and other information can be found at AVA primary site:

```
http://medo.fov.uni-mb.si/mapp
```

You can also subscribe to the MAPP ***AVR*** Mailing List

```
avr@medo.fov.uni-mb.si
```

To subscribe, send message to:

```
majordomo@medo.fov.uni-mb.si
```

with the following text in the body of the message:

```
subscribe avr your_email_address
```

Additional questions can be sent to my E-mail address:

```
uros.platise@ijs.si
```

B.2 Bug Report

Before reporting a bug please try to locate which part of the input code crashes down either assembler or linker. Send a detailed report to ***AVR*** Mailing List:

```
avr@medo.fov.uni-mb.si
```

or to my E-mail address:

```
uros.platise@ijs.si
```

Appendix C

Contact Addresses


```
Uros Platise
Seljakovo naselje 45
SI-4000 Kranj
Slovenia

telephone: +386 64 312 457
E-mail: uros.platise@ijs.si
```