AVR Assembler v. 2.0

Addendum to on-line help

Beta 3 – 2004-07-07

1 Introduction

This document explains the difference between the new AVR Assembler 2.0 (AVRASM2) and the current assembler (AVRASM) described in the AVR Studio on-line help.

This information will be added to the on-line help when AVRASM2 is officially released.

AVRASM2 is a complete re-write of the AVR Assembler.

It is fully backwards compatible with version 1.x (AVRASM), with a few minor exceptions that are detailed in chapter 2. It also offers a number of new powerful features, including:

- C-style preprocessing directives like `#define`, `#ifdef`, etc. This also includes the capability to control definition/undefinition of preprocessor symbols and inclusion of files from the command line.
- Improved error detection and diagnostic messages, including exceeding available SRAM and EEPROM memory ranges.
- Support for floating point constant expressions and conversion to fractional numbers as used with the FMUL/FMUS/FMULSU instructions.
- Resource use statistics, including instructions, registers, and memory.
- Externalizing of device support, meaning support for new devices may be added without having to upgrade the assembler executable.

AVRASM2 is intended to replace v. 1.x completely, but will be offered as an alternative during a test period, to allow a painless transition between the two versions.

1.1 Intended Audience

This document is written for existing users of AVR Studio 4/AVR Assembler, and assumes some familiarity with the AVR architecture and AVR Assembler 1.x (AVRASM1) syntax. The AVR instruction set and AVRASM1 syntax are documented in the AVR Studio Online Help.

1.2 Support

This is a BETA release, and as such not supported via the normal ATMEL support channels. However, users may send feedback and bug reports directly to the developers, using the e-mail address avrbeta@atmel.com.

The *AVR Studio 4 forum* at http://www.avrfreaks.com is also a suitable channel for providing feedback or discussing this release.

2 Package contents

The following files are found at the download site:

• `AvrAsm2-Addendum.pdf`	This file.
• `README-AvrAsm2.txt`	README file
• `AvrAssembler2.exe`	Installation program

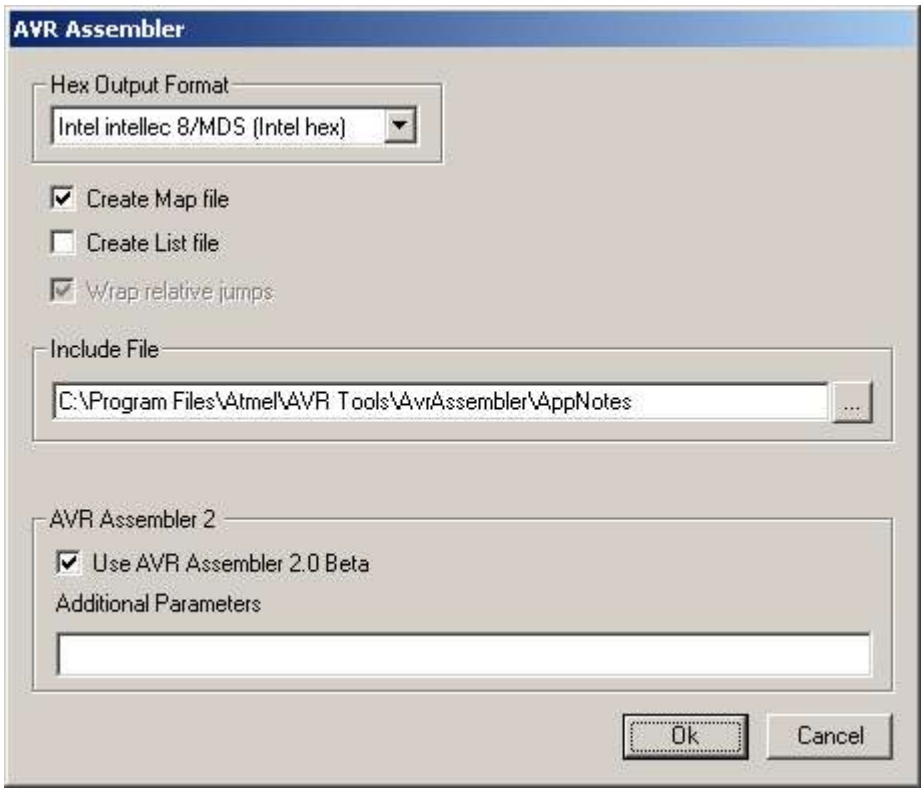
3 Installation

Download the file *AvrAssembler2.exe*, run it, and follow instructions.

The AVRASM2 beta assembler will be installed and enabled by default.

The assembler is configured by selecting *Project – AVR Assembler Setup* from the Studio menubar.

The upper part of this window is the same as for AVR Assembler 1. The lower part enables AVRASM2-Beta. The "Additional Parameters" field is used to add additional command-line parameters for AVRASM2. (Hint: Try adding **`-vs`** , see section 5 for explanation.)



3.1 Uninstallation

This add-on package cannot be uninstalled, but unselecting *Use AVR Assembler 2.0 beta* above will have the same effect, causing the standard 1.7x version to be used.

4 What's New

4.1 Changes in Beta 3

Bugs fixed:

- Macro argument expansion problem
- Preprocessor expression evaluation fails with multiple references to same macro.
- Preprocessor stack overflow caused by indirectly self-referencing macro.
- Comments in #define only work using C comments (Documentation issue)

4.2 Changes in Beta 2

Bugs fixed:

- byte directive not allowed in .eseg
- Missing/unclear error message when exceeding available SRAM/EEPROM
- Argument propagation in nested macro calls cause assembler to hang

5 Known issues

This chapter lists known issues in AVR Assembler 2.0 release BETA 1.

5.1 Broken include files and appnotes

The issues mentioned in this section will be fixed in a later AVRASM2 release.

AVRASM2 may produce one or both of the following warnings, depending on the device used:

```
tn2313def.inc(123): warning: Attempt to redefine keyword 'z', ignored
tn2313def.inc(383): warning: Attempt to redefine keyword 'or', ignored
```

These definitions - the Zero (Z) flag in the status register and a deprecated flag in the UCSRA register (use DOR instead of OR) will be renamed/removed from the .inc files.

As the warnings state, these definitions are *ignored* by AVRASM2. Any program not actually *using* these definitions will be fine.

Programs using these definitions cannot be assembled successfully, and will need to be changed.

The reasons for these warnings are explained in section 6.1.

5.2 Comments in macro calls

There is a known bug causing syntax errors in some situations when C-style comments (/* */, //) are used in lines with macro calls.

6 Invocation syntax

The invocation syntax is shown below, new options are **bold** and described below.

```
usage: avrasm2.exe [options] file.asm

 Options:
    -f [O|M|I|G|E] output file format:
        -fO Debug info for simulation in AVR Studio (default)
        -fM Motorola hex
        -fI Intel hex
        -fG Generic hex format
    -o ofile   Put output in 'ofile'.
    -d dfile   Generate debug info for simulation in AVR Studio in 'dfile'.
               Can only be used with the -f [M|I|G] option.
    -l lfile   Generate listing in 'lfile'
    -m mfile   Generate map in 'mfile'
    -e efile   Place EEPROM contents in 'efile'
    -w         Relative jumps are allowed to wrap for program ROM
               up to 4k words in size [ignored]
    -C ver     Specify AVR core version
    -c         Case sensitive
    -1/-2      Turn on/off AVR Assembler version 1 compatibility.
    -I dir     Preprocessor: Add 'dir' to include search path
    -i file    Preprocessor: Explicitly pre-include file
```

```
-D name[=value] Preprocessor: Define symbol. If =value is
          omitted, it is set to 1.
-U name    Preprocessor: Undefine symbol.
-v verbosity [0-9][s]:
    -vs Include target resource usage statistics
    -vl Output low-level assembly code to stdout
    -v0 Silent, only error messages printed
    -v1 Error and warning messages printed
    -v2 Error, warning, and  info messages printed (default)
    -v3-v9 Unspecified, increasing amounts of assembler internal dumps.
```

`-w`

Wrap relative jumps. This option is obsoleted. It is still recognized by the assembler, but ignored. The assembler will determine if wrapping is required based on flash memory size. If the flash size is unknown, wrap will be disabled.

`-C core-version`

Specify AVR Core version. The core version is normally specified in part definition files (*part*def..inc), this option is intended for testing of the assembler, and generally not useful for end-users.

`-c`

Causes the assembler to become entirely case sensitive. Preprocessor directives and macros are always case sensitive. Warning: Setting this option will break many existing projects.

`-1`
`-2`

Enable and disable AVRASM1 compatibility mode. As of BETA 1, this option is enabled by default (warning about this is printed). The default setting will be changed at a later time, and this option may be removed altogether. The compatibility mode will permit certain constructs otherwise considered errors, reducing the risk of breaking existing projects.

`-i file`

Include a file. `#include "file"` directive is processed before the first source code line is processed. Multiple –i directives may be used and are processed in order.

`-D name[=value]`
`-U name`

Define and undefine a preprocessor macro, respectively.
Note that function-type preprocessor macros may not be defined from the command line.

`-vs`

Print use statistics for register, instruction and memory on standard output. By default, only the memory statistic is printed. Note: The full statistics will always be printed to the list file, if one is specified.

`-vl`

This will print the raw instructions emitted to stdout, after all symbolic info is replaced. Mainly for assembler debugging purposes.

`-v0`

Print error messages only, warning and info messages are suppressed.

`-v1`
Print error and warning messages only, info messages are suppressed.

`-v2`
Print error, warning, and info messages. This is the default behaviour.

`-v3 ... -v9`
Add increasing amounts of assembler internal status dump. Mostly used for assembler debugging.

7 AVR Assembler 2 Syntax

7.1 Keywords

Unlike AVRASM1, keywords are reserved and cannot be redefined. The example below shows a program using keyword redefinitions that are illegal in AVRASM2.

EXAMPLE
The following nonsensical and rather obfuscated program assembles without errors with AVRASM1, but is incorrect with AVRASM2:

```
.equ    add = 2                 ; 'add' is a keyword (instruction)
.def    r1 = r16                ; 'r1' is a keyword (register)
.def    r0 = r31                ; 'r0' is a keyword (register)
.def    r16 = r24               ; 'r16' is a keyword (register)
.def    add = r2                ; 'add' is a keyword - again (instruction)
.def    mov = r29               ; 'mov' is a keyword (instruction)

mov:    cpi r16,100             ; 'mov' is a keyword - again (instruction)
        mov mov,mov             ; This gets interesting... :)
        ldi mov,mov             ; ... not to mention this!
        mov r1,add              ; Which definition of 'add' is used here?
        mov r0,r1               ; Things are not what they seem!
        rjmp mov                ; Now what...?
```

Assembler keywords are recognized regardless of case, unless the case-sensitive option is used (see section 5). If the case-sensitive option is used, assembler keywords will have the case shown here (generally lower case).

7.1.1 Instructions

```
adc     add     adiw    and     andi    asr
bclr    bld     brbc    brbs    brcc    brcs
break   breq    brge    brhc    brhs    brid
brie    brlo    brlt    brmi    brne    brpl
brsh    brtc    brts    brvc    brvs    bset
bst     call    cbi     cbr     clc     clh
cli     cln     clr     cls     clt     clv
clz     com     cp      cpc     cpi     cpse
dec     eicall  eijmp   elpm    eor     fmul
fmuls   fmulsu  icall   ijmp    in      inc
jmp     ld      ldd     ldi     lds     lpm
```

```
lsl     lsr     mov     movw    mul     muls
mulsu   neg     nop     or      ori     out
pop     push    rcall   ret     reti    rjmp
rol     ror     sbc     sbci    sbi     sbic
sbis    sbiw    sbr     sbrc    sbrs    sec
seh     sei     sen     ser     ses     set
sev     sez     sleep   spm     st      std
sts     sub     subi    swap    tst     wdr
```

7.1.2 Registers

```
r0   r1   r2   r3   r4   r5   r6   r7
r8   r9   r10  r11  r12  r13  r14  r15
r16  r17  r18  r19  r20  r21  r22  r23
r24  r25  r26  r27  r28  r29  r30  r31
x    y    z
```

7.1.3 Built-in functions and variables

```
abs     byte1    byte2    byte3    byte4
exp2    frac     high     hwrd     int
log2    low      lwrd     page     pc
q15     q7
```

7.1.4 Assembler directives

Assembler directives are recognized by the fact that they start with a period (‘.’). Any word preceded with a period will be attempted parsed as a directive.

There are no restrictions on the directive words without the leading period, nothing prevents the user from defining symbols like 'if' or 'byte'.

The list of directives is given here for the sake of completeness, not because the directives cause any problems:

```
.byte           .cseg           .csegsize       .db
.dd             .def            .device         .dq
.dseg           .dw             .elif           .else
.endif          .endm           .endmacro       .equ
.error          .eseg           .exit           .if
.ifdef          .ifndef         .include        .list
.listmac        .macro          .message        .nolist
.org            .set            .undef
```

8 Preprocessor

The AVRASM2 preprocessor is an integrated part of the assembler, handling C-style preprocessor directives and the corresponding macro expansion. A complete description of preprocessors is not given here, a good user guide for the GNU C preprocessor is found here:

http://www.delorie.com/gnu/docs/gcc/cpp_toc.html

The AVRASM2 preprocessor may largely be expected to work as described in this reference, unless otherwise explicitly stated.

Note: Preprocessor directives and macros are always case sensitive!

8.1 Preprocessor directives

The following preprocessor directives are implemented:

```
#define         #undef          #ifdef          #ifndef
#if             #else           #elif           #endif
#error          #include        #pragma         #
#warning        #message        defined
```

'#' above is the empty directive (does nothing). The `defined` keyword is only used in conjunction with `#if statements,` ie. `#if defined`

Not yet implemented:
`##` (concatenation) `#` (stringification)

Will not be implemented:
`#line`

Note
Only C-style comments (/* ... */ or // ...) are recognized by the preprocessor. Attempting to use assembler comment syntax (; ...) in preprocessor directives may give surprises.

8.2 Preprocessor directive overview

Only deviations from the C preprocessor are described here.

8.2.1 #define

```
1.#define name [value]
2.#define name(arg, ...) [value]
```

Note: Variadic macros (i.e., macros with variable number of arguments) are not implemented.

8.2.2 #undef

```
#undef name
```

8.2.3 #ifdef

```
#ifdef name
```

8.2.4 #ifndef

```
#ifndef name
```

8.2.5 #if/#elif

```
#if condition
#elif condition
```

The `defined(name)` operator is recognized in *condition.*

8.2.6 #else

```
#else
```

8.2.7 #endif

```
#endif
```

8.2.8 #error/#warning/#message

```
#error tokens
#warning tokens
#message tokens
```

Unlike the GNU C preprocessor, the `#error`, `#warning`, and `#message` directives will expand unquoted preprocessor macros. Example:

```
#message "Part name:" __PART_NAME__
```
may produce the output

```
Part name: Atmega48
```

8.2.9 #include

```
1.#include "file"
2.#include <file>
```

Note: Computed `#include` is not implemented (i.e., using a macro to specify an include file name)

8.2.10 #pragma

```
#pragma tokens
```

See description below.

8.2.11 # (empty directive)

```
#
```

8.3 Pragmas

Pragmas are used to specify part-specific properties, normally set up in include files. Normally there is no reason to use pragmas in user programs.

The supported pragmas are:

```
#pragma AVRPART ADMIN PART_NAME string
#pragma AVRPART CORE CORE_VERSION version-string
#pragma AVRPART CORE INSTRUCTIONS_NOT_SUPPORTED mnemonic[ operand[,operand] ][:...]
#pragma AVRPART CORE NEW_INSTRUCTIONS mnemonic[ operand[,operand]][:...]
```

```
#pragma AVRPART MEMORY PROG_FLASH size
#pragma AVRPART MEMORY EEPROM size
#pragma AVRPART MEMORY INT_SRAM SIZE size
#pragma AVRPART MEMORY INT_SRAM START_ADDR address
```

Notes
- The *string* values here are not quoted, and that the *numeric* values must be pure numbers (expressions or preprocessor macros not allowed here).
- Everything in the #pragma directive is case sensitive.
- Distinguishing between variants of the same instruction based on operands only works for instructions that are implemented with different operands in different variants on different AVR Core versions., presently these are the *ld*, *st*, and *lpm* instructions.

8.4 Pre-defined preprocessor macros

```
__AVRASM_VERSION__
__CORE_VERSION__
__DATE__
__FILE__
__LINE__
__PART_NAME__
__TIME__
```

Part-dependent:
__*partname*__ *partname* corresponds to the value of __PART_NAME__ above.

__CORE_*coreversion*__ *coreversion* corresponds to the value of __CORE_VERSION__ above.

9 Other syntactic elements

9.1 Comments

The following comment styles are recognized:

```
;   The rest of the line is a comment (classic assembler comments)
//  Like ';', the rest of the line is a comment
/*  Block comment; the enclosed text is a comment, may span multiple lines.
    This style of comments cannot be nested. */
```

9.1.1 Preprocessor note

The preprocessor only recognizes C-style comments. The semicolon (;) is not treated in any special way in preprocessor directives and this may have unexpected effects.

To avoid surprises, C-style comments should be used exclusively in lines with prepocessor directives.

Examle:

```
#define FOO 42 ; The foobaristic constant
#define BAR 43 // This is a bar, plain and simple
```

These definitions will result in FOO having the value "42 ; The foobaristic constant", while BAR simply has the value "43". This may or may not be a problem depending on how FOO is used.

9.2 User symbols

User symbols may consist of letters (a-z,A-Z), digits (0-9), and underscore (_). The first character cannot be a digit, and keywords are reserved.

9.3 Instructions

Instructions take the following forms:

```
mnemonic
mnemonic operand
mnemonic operand, operand2
```

Allowed operands are registers r0-r31, x, y, z, x+, y+, z+, -x, -y, -z, and integer expressions including preprocessor macros, constants defined with assembler directives, labels, and functions.

Multiple instructions in a single line is allowed, but not recommended. It is supported to faciliate preprocessor macros (see below).

9.4 Labels

A label is a user symbol followed by a colon (:). It may prepend an instruction or directive on the same line or be placed on a separate line.

9.5 Expressions

Constant expressions can be integer or floating point, and follow the C rules for operator precedence and type propagation. Symbols and instruction operands are always integer, a suitable conversion function should be used to convert before assignment (eg, int(), q7()).

Implicit float→int conversion cause a warning and the fractional part is discarded.

All C operators except '?:', '++', '--' are supported.

9.6 Strings and character constants

A string enclosed in double quotes (") can only be used in conjunction with the .db directive. The string is taken literally, no escape sequences are recognized, and it is not NUL-terminated.

Character constants can be used anywhere an integer expression is allowed, and the following C-style escape sequences are recognized, with the same meaning as in C:

`\n`	Newline (ASCII LF 0x0a)
`\r`	Carriage return (ASCII CR 0x0d)
`\a`	Alert bell (ASCII BEL 0x07)
`\b`	Backspace (ASCII BS 0x08)
`\f`	Form feed (ASCII FF 0x0c)
`\t`	Horizontal tab (ASCII HT 0x09)
`\v`	Vertical tab (ASCII VT 0x0b)
`\\`	Backslash
`\0`	Null character (ASCII NUL)

`\ooo` (*ooo* = octal number) and `\xhh` (*hh* = hex number) are also recognized.

Examples:

```
.db "Hello\n"  // is equivalent to:
.db 'H', 'e', 'l', 'l', 'o', '\\', 'n'
```

To create the equivalent to the C-string "Hello, world\n", do as follows:

```
.db "Hello, world", '\n', 0
```

9.7 Numeric constants

C-style integer constants: *dddd,* 0*oooo*, 0x*hhhh* are recognized as decimal, octal, and hexadecimal, respectively

Floating point constant formats are likewise according to C.

Additonally, the forms $*hhhh* and 0b*dddd* are recognized as hexadecimal and binary integers, respectively. These forms are also recognized and evaluated by the preprocessor.

10 Preprocessor and assembler interaction

Some of the assembler and preprocessor features have very similar functions (e.g. include, conditionals), and may interact in unexpected ways.

10.1 Include.

`#include` and `.include` are identical, except that `#include` also supports the '`#include <file>`' form.

10.2 Conditionals.

Preprocessor conditionals (#if, etc) relate to preprocessor symbols only (defined with #define or the -D command-line option).

Assembler conditional relate to both preprocessor and assembler symbols, but beware of unexpected effects:

```
#define FOO 2
...
.ifdef FOO
```

is a syntax error, because the assembler will try to evaluate '.ifdef 2'.

Avoid mixing the two forms! Preprocessor macros may be defined from the command line, and this gives considerable flexibility.

10.3 Macros

Both the preprocessor and the assembler offer macros, but they work very differently and should not be intermixed.

The traditional assembler macro

```
.macro  pushw
     push @0L
     push @0H
.endmacro
```

Each line of the macro body remains a separate line after expansion. Labels in the macro body have scope limited to the macro body.

Call of this macro looks like an instruction, e.g.:

```
     pushw z
```

Preprocessor function-style macros

A corresponding preprocessor macro would be

```
#define pushw(wreg) \
        push wreg##L \
        push wreg##H
```

(**Note:** the preprocessor concatenation operator ## is not yet implemented in BETA 1.)

Labels defined in a preprocessor macro have global scope, and the entire macro body becomes one single line after expansion.

It is called like a function, e.g.

```
   pushw(z)
```

It is generally not recommended to use preprocessor macros in this fashion. Preprocessor macros are best suited for implementing constant expressions, e.g.

```
#define square(x) (x)*(x)
```

10.4 Comments

The preprocessor only recognizes C-style comments (/*....*/, //....).
Assembler comments should not be used in conjunction with preprocessor directives. If assembler comments are used in conjunction with #define directives, the comment will become part of the macro, which may cause surprises when the macro is expanded.

11 Other new features

The following minor new features are also introduced.

11.1 Assembler directives

`.undef` *`symbol`*

Undefines a register definition created with the .def directive. Use this to avoid "*already defined by the .DEF directive*" warnings.

```
.dd expression[, expression ...]
.dq expression[, expression ...]
```

These directives are similar to the .dw directive, except they are used to define doublewords (32-bit) and quadwords (64-bit), respectively. The byte and word ordering is strictly little-endian.

11.2 New functions

`int()`
Truncates a floating point expression to integer (ie discards fractional part)

`frac()`
Extracts fractional part of a floating point expression (ie discards integer part).

`q7()`
Converts a fractional floating point expression to a form suitable for the FMUL/FMULS/FMULSU instructions. (sign + 7-bit fraction)

`q15()`
Converts a fractional floating point to the form returned by the FMUL/FMULS/FMULSY instructions (sign + 15-bit fraction).

`abs()`
Returns the absolute value of a constant expression.

11.3 Built-in "known place" for included files

Unlike AVRASM1, when AVRASM2 is installed as part of an AVR Studio installation, it will know where the Appnotes (include) directory is, and doesn't need to be told this via a command line option like

```
-I "C:\Program Files\Atmel\AVR Tools\AvrAssembler\Appnotes"
```

11.4 Motorola hex for large files

Motorola hex (S-record) output is now implemented for addresses above the 64kB limit.

11.5 Nested macro calls

Unlike AVRASM1, nested macro calls are supported. Use of this feature is *not* recommended, as it is very easy to lose track of what's going on.

Nested macro *definitions* are not allowed.