AVR Assembler

Preface

Welcome to the Microchip AVR® Assembler.

The Assembler generates fixed code allocations, consequently no linking is necessary.

The AVR Assembler is the assembler formerly known as AVR Assembler 2 (AVRASM2). The former AVRASM distributed with AVR Studio® 4 has now been obsoleted and will not be distributed with current products.

For documentation on the instruction set of the AVR family of microcontrollers, refer to the 8-bit AVR Instruction Set Manual.

Table of Contents

1. AVR Assembler Known Issues

Issue #4146: Line continuation doesn't work in macro calls

The following program illustrates this issue.

```
.macro m
ldi @0, @1
.endm m r16,\ 0
```

This is not a problem with preprocessor macros (`#define`).

Missing newline at end of file

AVRASM2 has some issues if the last line in a source file is missing a newline: Error messages may refer to wrong filename/line number if the error is in the last line of the included files, and in some cases syntax errors may result. Beware that the Atmel Studio editor will not append a missing newline at the end of a source file automatically.

Increment/decrement operators

Increment/decrement operators (++/--) are recognized by the assembler and may cause surprises, e.g.:

`symbol--1` will cause a syntax error, write `symbol - -1` if the intention is to subtract -1.

This behavior is consistent with C compilers. The ++/-- operators are not useful in the current assembler, but are reserved for future use.

Forward references in conditionals

Using a forward reference in an assembler conditional may cause surprises, and in some cases is not allowed. Example:

```
.org LARGEBOOTSTART
; the following sets up RAMPZ:Z to point to a FLASH data object, typically
; for use with ELPM.

ldi ZL, low (cmdtable * 2)

ldi ZH, high (cmdtable * 2)
.if ((cmdtable * 2) > 65535)

ldi r16, 1

sts RAMPZ, r16
.endif

; more code follows here
cmdtable: .db "foo", 0x0
```

The reason for this is that the outcome of the conditional will influence the value of the forward referenced label, which in turn may affect the outcome of the conditional, and so on.

The following is allowed:

```
.ifdef FOO
nop ; some code here
.endif
rjmp label ; more code here
.equ FOO = 100
label: nop
```

In this example FOO is not defined at the point it is used in a conditional. The use of .ifdef in this situation is allowed, and the conditional is false. However, the pattern shown above is not recommended because the programmer's intention is not clear. The form is intended to allow common constructs like this:

```
; Define FOO if it is not already defined.
.ifndef FOO
.equ FOO = 0x100
.endif
```

Up to and including AVRASM 2.0.30, these situations were not always properly detected, causing incomprehensible error messages. Starting with 2.0.31, explicit error messages are given.

Note that with preprocessor conditionals (`#if/#ifdef`), the situation is always well-defined, preprocessor symbols are always undefined until the definition is seen, and this kind of error will never occur.

Error messages

Sometimes error messages may be hard to understand. Typically, a simple typo in some instances may produce error messages like this:

```
myfile.asm(30): error: syntax error, unexpected FOO
```

where FOO represents some incomprehensible gibberish. The referenced filename/line number is correct, however.

***defined* incorrectly treated as an assembler keyword**

The keyword *defined* is recognized in all contexts. It should only be recognized in conditionals. This prevents *defined* to be used as a user symbol like a label, etc. On the other hand, it allows for constructs like '.dw foo = defined(bar)', which it shouldn't. Note that the preprocessor and assembler have separate implementations of *defined*. The exact behavior of *defined* currently (from 2.1.5) is:

- The preprocessor 'defined' keyword only relates to symbols defined with `#define`, and correctly does this only in preprocessor conditionals (`#if/#elif`).
- In all remaining code, the assembler's notion of *defined* is used, the correct behavior would be to only recognize it in assembler conditionals (`.if/.elif`).

Preprocessor issues

- The preprocessor will not detect invalid preprocessor directives inside a false conditional. This may lead to surprises with typos like this:

  ```
  #if __ATmega8__
  //...
  #elseif __ATmega16__ //WRONG, the correct directive is #elif
  // This will go undetected if __ATmega8__ is false
  //...
  #else
  // when __ATmega8__ is false this section will be assembled even if
  // __ATmega16__ is true.
  #endif
  ```

  It is debatable if this is a bug, the behavior is consistent with the C preprocessor.
- Issue #3361: The preprocessor incorrectly allows additional text after directives, which may cause surprises, e.g., #endif #endif will be interpreted as a single #endif directive, without any error or warning message.

- Issue #4741: Assembler conditionals in preprocessor macros don't work. Use of the macro defined below will result in different syntax error messages, depending on the value of the conditional val (true or false).

```
#define TEST \
.IF val \
.DW 0 \
.ELSE \
.DW 1 \
.ENDIF
```

  The reason for this is that assembler conditionals must appear on a separate line, and a preprocessor macro like the above is concatenated into a single line.

2. AVR Assembler Command Line Options

AVRASM2 may be used as a stand-alone program from the command line. The AVRASM2 command-line invocation syntax is shown below.

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

       Options:
-f [O|M|I|G|-] output file format:
-fO Debug info for simulation in Atmel Studio (default)
-fO1 | -fO2 - force format version 1 or 2 (default: auto)
-fM Motorola hex -fI Intel hex -fG Generic hex format
-f- No output file -o ofile Put output in 'ofile'.
-d dfile Generate debug info for simulation in Atmel 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. [Deprecated]
-p1|0 Set/unset AVRASM1 implicit .device include (also set by -1) [Deprecated]
-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.
-S file Produce include/label info file for Atmel Studio
-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.
-V  Format map and list files for Verilog.
-O i|w|e Overlap report: ignore|warning|error [error]
-W-b|+bo|+bi Byte operand out of range warning disable|overflow|integer
-W+ie|+iw Unsupported instruction error | warning
-W+fw  Label slip caused by forward ref accepted (DANGEROUS)
-FD|Tfmt __DATE__ | __TIME__ format, using strftime(3) format string
```

-f output-file-format

Supported formats are generic/Intel/Motorola hex, and AVR Object files. There are two sub variants of the AVR Object file format:

- Standard (V1) format, with 16-bit line number fields, supporting source files with up to 65534 lines
- Extended (V2) format, with 24-bit line number fields, supporting source files with up to ~16M lines

By default, when output format is unspecified or specified with -fO, the assembler will select the appropriate format automatically, V1 if the file has less than 65533 lines, V2 if it has more. The -fO1 and -fO2 options may be used to force V1 or V2 output file format regardless of number of lines.

If V1 file format is used with source files with more than 65534 lines, the assembler will issue a warning, and the lines above 65534 cannot be debugged with Atmel Studio.

For all normal assembler projects, the default option should be safe. The extended format is primarily intended for machine-generated assembly files.

-w

Wrap relative jumps. This option is obsoleted, because AVRASM2 automatically determines when to wrap relative jumps, based on program memory size. The option is recognized but ignored.

-C core-version

Specify AVR Core version. The core version is normally specified in part definition files (partdef.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 [Deprecated]

Enable and disable AVRASM1 compatibility mode. This mode is disabled (-2) by default. The compatibility mode will permit certain constructs otherwise considered errors, reducing the risk of breaking existing projects. It also affects the built-in include path, causing the assembler to look for device definition include files (devicedef.inc) for Assembler 1 in `C:\Atmel\AVR Tools\AvrAssembler\Appnotes` instead of the new Assembler 2 files in `C:\Atmel\AVR Tools\AvrAssembler2\Appnotes`.

Note: 1 option is deprecated and will not work in Atmel Studio 6 because the AVRASM1 include files are no longer distributed.

-I directory

Add directory to the include file search path. This affects both the preprocessor #include directive and the assembler INCLUDE directive.

Multiple -I directives may be given. Directories are searched in the order specified.

-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 will be 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. If -D is given no value, it is set to 1.

-S

Produces information about the include files, output files and label information Contained in the source file.

-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 standard output, 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 behavior.

-v3 ... -v9

Print increasing amounts of assembler internal status dump. Mostly used for assembler debugging.

-V

Formats the List and Map File for Verilog. It sets the Verilog Option.

-O i|w|e

If different sections of code are mapped to overlapping memory locations using the ORG directive, an error message is normally issued.

This option allows setting this condition to cause an error (-Oe, default), a warning (-Ow) or be completely ignored (-Oi). Not recommended for normal programs.

This may also be set by #pragma overlap directive.

-W-b |-W+bo | -W+bi

-b, +bo, and +bi correspond to no warning, warning when overflow, and warning when integer value out of range, respectively. This may also be set by #pragma warning range.

-W+ie|+iw

+ie and +iw selects if use of unsupported instructions gives error or warning, respectively. The default is to give an error. Corresponds to #pragma error instruction/pragma warning instruction, respectively.

-FDformat -FTformat

Specify the format of the __DATE__ and __TIME__ Pre-defined Macros, respectively. The format string is passed directly to the strftime(3) C library function. The __DATE__ and __TIME__ preprocessor macros will always be string tokens, i.e., their values will appear in double quotes.

The default formats are "%b %d %Y" and "%H:%M:%S", respectively.

Example: To specify ISO format for __DATE__, specify `-FD"%Y-%m-%d"`

These formats may only be specified at the command line, there are no corresponding #pragma directives.

Note: The Windows® command interpreter (cmd.exe or command.com) may interpret a character sequence starting and ending with a '%' character as an environment variable to be expanded even when it is quoted. This may cause the date/time format strings to be changed by the command interpreter and not work as expected. A workaround that will work in many cases is to use double '%' characters to specify the format directives, e.g., `-FD"%%Y-%%m-%%d"`for the example above. The exact behavior of the command interpreter seems to be inconsistent and vary depending on a number of circumstances, for one, it is different in batch and interactive mode. The effect of the format directives should be tested. It is recommended to put the following line in the source file for testing this:

```
#message "__DATE__ =" __DATE__ "__TIME__ =" __TIME__
```

This will print the value of the date and time macros when the program is assembled, making verification easy (see #error, #warning, and #message directive documentation). An alternative syntax for the format specification may be considered in future AVRASM2 versions to avoid this problem.

Some relevant strftime() format specifiers (see strftime(3) manual page for full details):

- %Y - Year, four digits
- %y - Year, two digits
- %m - Month number (01-12)
- %b - Abbreviated month name
- %B - Full month name
- %d - Day number in month (01-31)
- %a - Abbreviated weekday name
- %A - Full weekday name
- %H - Hour, 24-hour clock (00-23)
- %I - Hour, 12-hour clock (01-12)
- %p - "AM" or "PM" for 12-hour clock
- %M - Minute (00-59)
- %S - Second (00-59)

3. Assembler Source

The Assembler works on source files containing instruction mnemonics, labels, and directives. The instruction mnemonics and the directives often take operands. Code lines should be limited to 120 characters.

Every input line can be preceded by a label, which is an alphanumeric string terminated by a colon. Labels are used as targets for jump and branch instructions and as variable names in Program memory and RAM.

In input line may take one of the four following forms:

`[label:] instruction [operands] [Comment]`

`[label:] directive [operands] [Comment]`

`Comment`

`Empty line`

A comment has the following form:

`; [Text]`

Items placed in braces are optional. The text between the comment-delimiter (;) and the end of line (EOL) is ignored by the Assembler. Labels, instructions and directives are described in more detail later. See also AVR Assembler Syntax.

Examples:

```
label:  .EQU var1=100 ; Set var1 to 100 (Directive)
        .EQU var2=200 ; Set var2 to 200

test:   rjmp test ; Infinite loop (Instruction)
        ; Pure comment line

        ; Another comment line
```

4. AVR Assembler Syntax

4.1 Keywords

Predefined identifiers (keywords) are reserved and cannot be redefined. The keywords include all Instruction mnemonics and Functions.

Assembler keywords are recognized regardless of case, unless the `-c` option is used, in which case the keywords are lower case (i.e., "add" is reserved, "ADD" is not).

4.2 Preprocessor Directives

AVRASM2 considers all lines starting with a '#' as the first non-space character a preprocessor directive.

4.3 Comments

In addition to the classic assembler comments starting with ';', AVRASM2 recognizes C-style comments. The following comment styles are recognized:

```
; The rest of the line is a comment (classic assembler comment)
// 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. */
```

4.4 Line Continuation

Like in C, source lines can be continued by means of having a backslash (\) as the last character of a line. This is particularly useful when defining long preprocessor macros, and for long `.db` directives.

```
.db 0, 1, "This is a long string", '\n', 0, 2, \
"Here is another one", '\n', 0, 3, 0
```

4.5 Integer Constants

AVRASM2 allows underscores (_) to be used as separators for increased readability. Underscores may be located anywhere in the number except as the first character or inside the radix specifier.

Example: `0b1100_1010` and `0b_11_00_10_10_` are both legal, while `_0b11001010` and `0_b11001010` are not.

4.6 Strings and Character Constants

A string enclosed in double quotes (") can be used only in conjunction with the DB directive and the MESSAGE/WARNING/ERROR directives. The string is taken literally, no escape sequences are recognized, and it is not NULL-terminated.

Quoted strings may be concatenated according to the ANSI C convention, i.e., `"This is a " "long string"` is equivalent to `"This is a long string"`. This may be combined with Line Continuation to form long strings spanning multiple source lines.

Character constants are enclosed in single quotes ('), and can be used anywhere an integer expression is allowed. The following C-style escape sequences are recognized, with the same meaning as in C:

Escape sequence	Meaning
\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'
.db '\0', '\177', '\xff'
```

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

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

4.7 Multiple Instructions per Line

AVRASM2 allows multiple instructions and directives per line, but its use is not recommended. It is needed to support expansion of multiline preprocessor macros.

4.8 Operands

AVRASM2 has support for integer operands and limited support for floating point constant expressions. All operands are described in the Operands section later in this user guide.

5. Assembler Directives

5.1 BYTE

Reserves bytes for a variable.

The BYTE directive reserves memory resources in the SRAM or EEPROM. In order to be able to refer to the reserved location, the BYTE directive should be preceded by a label. The directive takes one parameter, which is the number of bytes to reserve. The directive can not be used within a Code segment (see directives ESEG, CSEG, and DSEG). Note that a parameter must be given. The allocated bytes are not initialized.

Syntax

```
LABEL: .BYTE expression
```

Example

```
.DSEG
var1:  .BYTE 1 ; reserve 1 byte to var1
table: .BYTE tab_size ; reserve tab_size bytes

.CSEG
ldi r30,low(var1) ; Load Z register low
ldi r31,high(var1) ; Load Z register high
ld r1,Z ; Load VAR1 into register 1
```

5.2 CSEG

Code segment.

The CSEG directive defines the start of a Code Segment. An Assembler file can consist of several Code Segments, which are concatenated into one Code Segment when assembled. The BYTE directive can not be used within a Code Segment. The default segment type is Code. The Code Segments have their own location counter which is a word counter. The ORG directive can be used to place code and constants at specific locations in the Program memory. The directive does not take any parameters.

Syntax

```
.CSEG
```

Example

```
.DSEG ; Start data segment
vartab: .BYTE 4 ; Reserve 4 bytes in SRAM

.CSEG ; Start code segment
const: .DW 2 ; Write 0x0002 in prog.mem.
mov r1,r0 ; Do something
```

5.3 CSEGSIZE

Program Memory Size.

AT94K devices have a user configurable memory partition between the AVR Program memory and the data memory. The program and data SRAM is divided into three blocks: 10K x 16 dedicated program SRAM, 4K x 8 dedicated data SRAM, and 6K x 16 or 12K x 8 configurable SRAM, which may be

swapped between program and data memory spaces in 2K x 16 or 4K x 8 partitions. This directive is used to specify the size of the program memory block.

Syntax

```
.CSEGSIZE = 10 | 12 | 14 | 16
```

Example

```
.CSEGSIZE = 12 ; Specifies the program meory size as 12K x 16
```

5.4 DB

Define constant byte(s) in program memory and EEPROM.

The DB directive reserves memory resources in the program memory or the EEPROM memory. In order to be able to refer to the reserved locations, the DB directive should be preceded by a label. The DB directive takes a list of expressions, and must contain at least one expression. The DB directive must be placed in a Code Segment or an EEPROM Segment.

The expression list is a sequence of expressions, delimited by commas. Each expression must evaluate to a number between -128 and 255. If the expression evaluates to a negative number, the 8-bits twos complement of the number will be placed in the program memory or EEPROM memory location.

If the DB directive is given in a Code Segment and the expression-list contains more than one expression, the expressions are packed so that two bytes are placed in each program memory word. *If the expression-list contains an odd number of expressions, the last expression will be placed in a program memory word of its own, even if the next line in the assembly code contains a DB directive.* The unused half of the program word is set to zero. A warning is given, in order to notify the user that an extra zero byte is added to the .DB statement.

Syntax

```
LABEL: .DB expressionlist
```

Example

```
.CSEG
consts: .DB 0, 255, 0b01010101, -128, 0xaa

.ESEG
const2: .DB 1,2,3
```

5.5 DD

Define constant double-word(s) in program memory and EEPROM.

This directive is very similar to the DW directive, except it is used to define 32-bit (double-word). The data layout in memory is strictly little-endian.

Syntax

```
LABEL: .DD expressionlist
```

Example

```
.CSEG
varlist: .DD 0, 0xfadebabe, -2147483648, 1 << 30
```

5.6 DEF

Set a symbolic name on a register.

The DEF directive allows the registers to be referred to through symbols. A defined symbol can be used in the rest of the program to refer to the register it is assigned to. A register can have several symbolic names attached to it. A symbol can be redefined later in the program.

Syntax

```
.DEF Symbol=Register
```

Example

```
.DEF temp=R16
.DEF ior=R0

.CSEG
ldi temp,0xf0 ; Load 0xf0 into temp register
in ior,0x3f ; Read SREG into ior register
eor temp,ior ; Exclusive or temp and ior
```

5.7 DQ

Define constant quad-word(s) in program memory and EEPROM.

This directive is very similar to the DW directive, except it is used to define 64-bit (quad-word). The data layout in memory is strictly little-endian.

Syntax

```
LABEL: .DQ expressionlist
```

Example

```
.ESEG
eevarlst: .DQ 0,0xfadebabedeadbeef, 1 << 62
```

5.8 DSEG

Data Segment.

The DSEG directive defines the start of a Data segment. An assembler source file can consist of several data segments, which are concatenated into a single data segment when assembled. A data segment will normally consist of BYTE directives (and labels) only. The Data Segments have their own location counter which is a byte counter. The ORG directive can be used to place the variables at specific locations in the SRAM. The directive does not take any parameters.

Syntax

```
.DSEG
```

Example

```
.DSEG ; Start data segment
var1: .BYTE 1 ; reserve 1 byte to var1
table: .BYTE tab_size ; reserve tab_size bytes.

.CSEG
ldi r30,low(var1) ; Load Z register low
ldi r31,high(var1) ; Load Z register high
ld r1,Z ; Load var1 into register 1
```

5.9 DW

Define constant word(s) in program memory and EEPROM.

The DW directive reserves memory resources in the program memory or the EEPROM. In order to be able to refer to the reserved locations, the DW directive should be preceded by a label. The DW directive takes a list of expressions, and must contain at least one expression. The DB directive must be placed in a Code Segment or an EEPROM Segment.

The expression list is a sequence of expressions, delimited by commas. Each expression must evaluate to a number between -32768 and 65535. If the expression evaluates to a negative number, the 16-bits two's complement of the number will be placed in the program memory or EEPROM location.

Syntax

```
LABEL: .DW expressionlist
```

Example

```
.CSEG
varlist: .DW 0, 0xffff, 0b1001110001010101, -32768, 65535

.ESEG
eevarlst: .DW 0,0xffff,10
```

5.10 ELIF and ELSE

Conditional assembly.

.ELIF will include code until the corresponding ENDIF of the next ELIF at the same level if the expression is true, and both the initial .IF clause and all following .ELIF clauses are false.

.ELSE will include code until the corresponding .ENDIF if the initial .IF clause and all .ELIF clauses (if any) all are false.

Syntax

```
.ELIF <expression>
.ELSE
.IFDEF <symbol> |.IFNDEF <symbol>
…
.ELSE | .ELIF <expression>
…
```

Example

```
.IFDEF DEBUG
.MESSAGE "Debugging.."
.ELSE
.MESSAGE "Release.."
.ENDIF
```

5.11 ENDIF

Conditional assembly.

Conditional assembly includes a set of commands at assembly time. The ENDIF directive defines the end for the conditional IF, IFDEF, or IFNDEF directives.

Conditionals (.IF...ELIF...ELSE...ENDIF blocks) may be nested, but all conditionals must be terminated at the end of file (conditionals may not span multiple files).

Syntax

```
.ENDIF
.IFDEF <symbol> |.IFNDEF <symbol>
…
.ELSE | .ELIF <expression>
…
.ENDIF
```

Example

```
.IFNDEF DEBUG
.MESSAGE "Release.."
.ELSE
.MESSAGE "Debugging.."
.ENDIF
```

5.12 ENDM and ENDMACRO

End macro.

The ENDMACRO directive defines the end of a macro definition. The directive does not take any parameters. See the MACRO directive for more information on defining macros. ENDM is an alternative form, fully equivalent with ENDMACRO.

Syntax

```
.ENDMACRO
.ENDM
```

Example

```
.MACRO SUBI16 ; Start macro definition
subi r16,low(@0) ; Subtract low byte
sbci r17,high(@0) ; Subtract high byte
.ENDMACRO
```

5.13 EQU

Set a symbol equal to an expression.

The EQU directive assigns a value to a label. This label can then be used in later expressions. A label assigned to a value by the EQU directive is a constant and can not be changed or redefined.

Syntax

```
.EQU label = expression
```

Example

```
.EQU io_offset = 0x23
.EQU porta = io_offset + 2.CSEG ; Start code segment
clr r2 ; Clear register 2
out porta,r2 ; Write to Port A
```

5.14 ERROR

Outputs an error message string.

The ERROR directive outputs a string and halts the assembling. May be used in conditional assembly.

Syntax

```
.ERROR "<string>"
```

Example

```
.IFDEF TOBEDONE
.ERROR "Still stuff to be done.."
.ENDIF
```

5.15 ESEG

EEPROM Segment.

The ESEG directive defines the start of an EEPROM segment. An assembler source file can consist of several EEPROM segments, which are concatenated into a single EEPROM segment when assembled. An EEPROM segment will normally consist of DB and DW directives (and labels) only. The EEPROM segments have their own location counter, which is a byte counter. The ORG directive can be used to place the variables at specific locations in the EEPROM. The directive does not take any parameters.

Syntax

```
.ESEG
```

Example

```
.DSEG ; Start data segment
var1: .BYTE 1 ; reserve 1 byte to var1
table: .BYTE tab_size ; reserve tab_size bytes.

.ESEG
eevar1: .DW 0xffff ; initialize 1 word in EEPROM
```

5.16 EXIT

Exit this file.

The EXIT directive tells the Assembler to stop assembling the file. Normally, the Assembler runs until end of file (EOF). If an EXIT directive appears in an included file, the Assembler continues from the line following the INCLUDE directive in the file containing the INCLUDE directive.

Syntax

```
.EXIT
```

Example

```
.EXIT ; Exit this file
```

5.17 IF, IFDEF, and IFNDEF

Conditional assembly.

Conditional assembly includes a set of commands at assembly time. The IFDEF directive will include code till the corresponding ELSE directive if <symbol> is defined. The symbol must be defined with the EQU or SET directive. (Will not work with the DEF directive.) The IF directive will include code if the <expression> is evaluated different from 0. Valid till the corresponding ELSE or ENDIF directive.

Up to five levels of nesting is possible.

Syntax

```
.IFDEF <symbol>
.IFNDEF <symbol>
.IF <expression>
```

```
.IFDEF <symbol> |.IFNDEF <symbol>
…
.ELSE | .ELIF <expression>
…
.ENDIF
```

Example

```
.MACRO SET_BAT
.IF @0>0x3F
.MESSAGE "Address larger than 0x3f"
lds @2, @0
sbr @2, (1<<@1)
sts @0, @2
.ELSE
.MESSAGE "Address less or equal 0x3f"
.ENDIF
.ENDMACRO
```

5.18 INCLUDE

Include another file.

The INCLUDE directive tells the Assembler to start reading from a specified file. The Assembler then assembles the specified file until end of file (EOF) or an EXIT directive is encountered. An included file may contain INCLUDE directives itself. The difference between the two forms is that the first one search the current directory first, the second one does not.

Syntax

```
.INCLUDE "filename"
.INCLUDE <filename>
```

Example

```
; iodefs.asm:
.EQU sreg = 0x3f ; Status register
.EQU sphigh = 0x3e ; Stack pointer high
.EQU splow = 0x3d ; Stack pointer low

; incdemo.asm
.INCLUDE iodefs.asm ; Include I/O definitions
in r0,sreg ; Read status register
```

5.19 LIST

Turn the listfile generation on.

The LIST directive tells the Assembler to turn listfile generation on. The Assembler generates a listfile, which is a combination of assembly source code, addresses, and opcodes. Listfile generation is turned on by default. The directive can also be used together with the NOLIST directive in order to only generate listfile of selected parts of an assembly source file.

Syntax

```
.LIST
```

Example

```
.NOLIST ; Disable listfile generation
.INCLUDE "macro.inc" ; The included files will not
.INCLUDE "const.def" ; be shown in the listfile
.LIST ; Reenable listfile generation
```

5.20 LISTMAC

Turn macro expansion on.

The LISTMAC directive tells the Assembler that when a macro is called, the expansion of the macro is to be shown on the listfile generated by the Assembler. The default is that only the macro-call with parameters is shown in the listfile.

Syntax

```
.LISTMAC
```

Example

```
.MACRO MACX ; Define an example macro
add r0,@0 ; Do something
eor r1,@1 ; Do something
.ENDMACRO ; End macro definition

.LISTMAC ; Enable macro expansion

MACX r2,r1 ; Call macro, show expansion
```

5.21 MACRO

Begin macro.

The MACRO directive tells the Assembler that this is the start of a Macro. The MACRO directive takes the Macro name as parameter. When the name of the Macro is written later in the program, the Macro definition is expanded at the place it was used. A Macro can take up to 10 parameters. These parameters are referred to as @0-@9 within the Macro definition. When issuing a Macro call, the parameters are given as a comma separated list. The Macro definition is terminated by an ENDMACRO directive.

By default, only the call to the Macro is shown on the listfile generated by the Assembler. In order to include the macro expansion in the listfile, a LISTMAC directive must be used. A macro is marked with a + in the opcode field of the listfile.

Syntax

```
.MACRO macroname
```

Example

```
.MACRO SUBI16 ; Start macro definition
subi @1,low(@0) ; Subtract low byte
sbci @2,high(@0) ; Subtract high byte
.ENDMACRO ; End macro definition

.CSEG ; Start code segment
SUBI16 0x1234,r16,r17 ; Sub.0x1234 from r17:r16
```

5.22 MESSAGE

Output a message string.

The MESSAGE directive outputs a string. Useful in conditional assembly.

Syntax

```
.MESSAGE "<string>"
```

Example

```
.IFDEF DEBUG
.MESSAGE "Debug mode"
.ENDIF
```

5.23 NOLIST

Turn listfile generation OFF.

The NOLIST directive tells the Assembler to turn listfile generation OFF. The Assembler normally generates a listfile, which is a combination of assembly source code, addresses, and opcodes. Listfile generation is turned on by default, but can be disabled by using this directive. The directive can also be used together with the LIST directive in order to only generate listfile of selected parts of an assembly source file.

Syntax

```
.NOLIST
```

Example

```
.NOLIST ; Disable listfile generation
.INCLUDE "macro.inc" ; The included files will not
.INCLUDE "const.def" ; be shown in the listfile
.LIST ; Reenable listfile generation
```

5.24 ORG

Set program origin.

The ORG directive sets the location counter to an absolute value. The value to set is given as a parameter. If an ORG directive is given within a Data Segment, then it is the SRAM location counter which is set, if the directive is given within a Code Segment, then it is the Program memory counter which is set and if the directive is given within an EEPROM Segment, it is the EEPROM location counter which is set.

The default values of the Code and the EEPROM location counters are zero, and the default value of the SRAM location counter is the address immediately following the end of I/O address space (0x60 for devices without extended I/O, 0x100 or more for devices with extended I/O) when the assembling is started. Note that the SRAM and EEPROM location counters count bytes whereas the Program memory location counter counts words. Also note that some devices lack SRAM and/or EEPROM.

Syntax

```
.ORG expression
```

Example

```
.DSEG ; Start data segment
.ORG 0x120; Set SRAM address to hex 120
variable: .BYTE 1 ; Reserve a byte at SRAM adr. 0x120

.CSEG
.ORG 0x10 ; Set Program Counter to hex 10
mov r0,r1 ; Do something
```

5.25 OVERLAP and NOOVERLAP

Set up overlapping section.

Introduced in AVRASM 2.1. These directives are for projects with special needs and should normally not be used.

These directives only affect the currently active segment (CSEG, DSEG, and ESEG).

The .overlap/nooverlap directives mark a section that will be allowed to overlap code/data with code/data defined elsewhere, without any error or warning messages being generated. This is totally independent of what is set using the #pragma, General Purpose directives. The overlap-allowed attribute will stay in effect across .org directives, but will not follow across .cseg/.eseg/.dseg directives (each segment marked separately).

The typical use of this is to set up some form of default code or data that may or may not later be modified by overlapping code or data, without having to disable assembler overlap detection altogether.

Syntax

```
.OVERLAP
.NOOVERLAP
```

Example

```
.overlap
.org 0 ; section #1
rjmp default
.nooverlap
.org 0 ; section #2
rjmp RESET ; No error given here
.org 0 ; section #3
rjmp RESET ; Error here because overlap with #2
```

5.26 SET

Set a symbol equal to an expression.

The SET directive assigns a value to a label. This label can then be used in later expressions. Unlike the EQU directive, a label assigned to a value by the SET directive can be changed (redefined) later in the program.

Syntax

```
.SET label = expression
```

Example

```
.SET FOO = 0x114; set FOO to point to an SRAM location
lds r0, FOO; load location into r0
.SET FOO = FOO + 1 ; increment (redefine) FOO. This would be illegal if using .EQU
lds r1, FOO ; load next location into r1
```

5.27 UNDEF

Undefine a register symbolic name.

The UNDEF directive is used to undefine a symbol previously defined with the DEF directive. This provides a way to obtain a simple scoping of register definitions, to avoid warnings about register reuse.

Syntax

```
.UNDEF symbol
```

Example

```
.DEF var1 = R16
ldi var1, 0x20
... ; do something more with var1
.UNDEF var1

.DEF var2 = R16 ; R16 can now be reused without warning.
```

5.28 WARNING

Outputs a warning message string.

The .WARNING directive outputs a warning string, but unlike the .ERROR directive it does not halt assembling. May be used in conditional assembly.

Syntax

```
.WARNING"<string>"
```

Example

```
.IFDEF EXPERIMENTAL_FEATURE
.WARNING "This is not properly tested, use at own risk."
.ENDIF
```

6. Preprocessor

The AVRASM2 preprocessor is modeled after the C preprocessor, with some exceptions:

- It recognizes all integer formats used by AVRASM2, i.e., `$abcd` and `0b011001` are recognized as valid integers by the preprocessor and can be used in expressions in `#if` directives.
- `'.'` and `'@'` are allowed in identifiers. `'.'` is required to allow preprocessor directives like `'.dw'` to be used in preprocessor macros, `'@'` is required to handle assembler macro arguments correctly.
- It recognizes assembler-style comment delimiters (`';'`) as well as C-style comments. Using `';'` as a comment delimiter is in conflict with the C use of `';'`, and it is therefore not recommended to use assembler-style comments in conjunction with preprocessor directives.
- The `#line` directive is not implemented
- Variadic macros (i.e., macros with variable number of arguments) are not implemented
- The `#warning` and `#message` directives are not specified in the ANSI C standard

6.1 #define

Syntax:

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

Description:

Define a preprocessor macro. There are two forms of macros; (1) object-like macros that basically define a constant, and (2) function-like macros that do parameter substitution.

```
value
```

may be any string, it is not evaluated until the macro is expanded (used). If `value` is not specified, it is set to 1.

Form (1) macros may be defined from the command line, using the `-D` argument.

When form (2) is used, the macro must be called with the same number of arguments it is defined with. Any occurrences of `arg` in `value` will be replaced with the corresponding `arg` when the macro is called. Note that the left parenthesis must appear immediately after `name` (no spaces between), otherwise it will be interpreted as part of the value of a form (1) macro.

Examples

Note that the placement of the first '(' is very significant in the examples below.

1. ```
   #define EIGHT (1 << 3)
   ```
2. ```
   #define SQR(X) ((X)*(X))
   ```

6.2 #undef

Syntax

`#undef name`

Description

Undefine macro name previously defined with a `#define` directive. If `name` is not previously defined, the `.undef` directive is silently ignored. This behavior is in accordance with the ANSI C standard. Macros may also be undefined from the command line using the `-U` argument.

6.3 #ifdef

Syntax

```
#ifdef name
```

Description

All the following lines until the corresponding `#endif`, `#else`, or `#elif` are conditionally assembled if `name` is previously `#defined`. Shorthand for `#if defined (name)`.

Example

```
#ifdef FOO
// do something
#endif
```

6.4 #ifndef

Syntax

```
#ifndef name
```

Description

The opposite of `#ifdef`: All following lines until the corresponding `#endif`, `#else`, or `#elif` are conditionally assembled if `name` is not `#defined`. Shorthand for `#if !defined (name)`.

6.5 #if and #elif

Syntax

```
#if condition
```

```
#elif condition
```

Description

All the following lines until the corresponding `#endif`, `#else`, or `#elif` are conditionally assembled if condition is true (not equal to 0). Condition may be any integer expression, including preprocessor macros, which are expanded. *The preprocessor recognizes the special operator `#defined (name)` that returns 1 if the name is defined and 0 otherwise. Any undefined symbols used in condition are silently evaluated to 0.*

Conditionals may be nested to arbitrary depth.

`#elif` evaluates condition in the same manner as `#if`, except that it is only evaluated if no previous branch of a compound `#if … #elif` sequence has been evaluated to true.

Examples

```
#if 0
// code here is never included
```

```
#endif

#if defined(__ATmega48__) || defined(__ATmega88__)
// code specific for these devices
#elif defined (__ATmega169__)
// code specific for ATmega169
#endif // device specific code
```

6.6 #else

Syntax

`#else`

Description

All the following lines until the corresponding `#endif` are conditionally assembled if no previous branch in a compound `#if` … `#elif` … sequence has been evaluated to true.

Example

```
#if defined(__ATmega48__) || defined(__ATmega88__)
// code specific for these parts
#elif defined (__ATmega169__)
// code specific for ATmega169
#else
#error "Unsupported part:" __PART_NAME__
#endif // part specific code
```

6.7 #endif

Syntax

`#endif`

Description

Terminates a conditional block initiated with an `#if`, `#ifdef`, or `#ifndef` directive.

6.8 #error, #warning, and #message

Syntax

`#error tokens`

`#warning tokens`

`#message tokens`

Description

`#error` emits `tokens` to standard error, and increments the assembler error counter, hereby preventing the program from being successfully assembled. `#error` is specified in the ANSI C standard.

`#warning` emits `tokens` to standard error, and increments the assembler warning counter. `#warning` is not specified in the ANSI C standard, but is commonly implemented in preprocessors such as the GNU C preprocessor.

`#message` emits `tokens` to standard output, and does not affect assembler error or warning counters. `#message` is not specified in the ANSI C standard.

For all directives, the output will include file name and line number, like normal error and warning messages.

`tokens` is a sequence of preprocessor `tokens`. Preprocessor macros are expanded except if appearing inside quoted strings `(")`.

Example

```
#error "Unsupported part:" __PART_NAME__
```

6.9 #include

Syntax

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

Description

Include a file. The two forms differ in that (1) searches the current working directory first, and is functionally equivalent with the assembler INCLUDE directive. (2) does not search the current working directory unless explicitly specifying it with an "." entry in the include path. Both forms will search a built-in known place after any explicitly specified path. This is the location for the partdef.inc include files supplied with the assembler.

It is strongly discouraged to use absolute path-names in `#include` directives, as this makes it difficult to move projects between different directories/computers. Use the `-I` argument to specify the include path, or set it up in **Atmel Studio => Project => Assembler Options**.

Examples

```
#include <m48def.inc>
#include "mydefs.inc"
```

6.10 #pragma, General Purpose

Syntax

1. `#pragma warning range byte option`
2. `#pragma overlap option`
3. `#pragma error instruction`
4. `#pragma warning instruction`

Description

1. The assembler evaluates constant integer expressions as 64-bit signed integers internally. When such expressions are used as immediate operands, they must be truncated to the number of bits required by the instructions. For most operands, an out-of-range value will cause an "operand out of range" error message. However, the immediate byte operands for the `LDI`, `CPI`, `ORI`, `ANDI`, `SUBI` and `SBCI` instructions (see also http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf) have several possible interpretations that are affected by this option.
   - **option = integer** The immediate operand is evaluated as an integer, and if its value is outside the range [-128 ... 255] a warning is given. The assembler doesn't know if the users intends

an integer operand to be signed or unsigned, hence it allows any signed or unsigned value that fits into a byte.

- **option = overflow (default)** The immediate operand is basically evaluated as an unsigned byte, and any sign extension bits are ignored. This option is particularly suitable when dealing with bit masks, when the integer interpretation would cause lots of warnings, like `ldi r16, ~((1 << 7) | (1 << 3))`
- **option = none** Disables all out-of-range warnings for byte operands. Not recommended.

2. If different sections of code are mapped to overlapping memory locations using the ORG directive, an error message is normally issued. This options modifies that behavior as follows:
   - **option = ignore** Ignores overlap conditions altogether; no errors, no warnings. Not recommended.
   - **option = warning** Produce warnings when overlaps are detected.
   - **option = error** Consider overlaps as error condition; this is the default and recommended setting.
   - **option = default** Revert to default handling -error or whatever is specified with the `-O` command line option.

   The assembler maintains two settings for overlap handling; The default setting that is set up by the `-O` command-line option, and the effective setting, that only can be modified with this `#pragma`. The two settings are equal upon assembler invocation. This `#pragma` changes the effective setting from the line it is invoked until the line it is changed by another invocation of this `#pragma`. Hence, this `#pragma` covers source line ranges and not address ranges. See also: OVERLAP and NOOVERLAP.
3. Causes use of instructions that are unsupported on the specified device to cause an assembler error (default behavior).
4. Causes use of instructions that are unsupported on the specified device to cause an assembler warning.

6.11 #pragma, AVR Part Related

Syntax

1. `#pragma AVRPART ADMIN PART_NAME string`
2. `#pragma AVRPART CORE CORE_VERSION version-string`
3. `#pragma AVRPART CORE INSTRUCTIONS_NOT_SUPPORTED mnemonic[ operand[,operand] ][:...]`
4. `#pragma AVRPART CORE NEW_INSTRUCTIONS mnemonic[ operand[,operand]][:...]`
5. `#pragma AVRPART MEMORY PROG_FLASH size`
6. `#pragma AVRPART MEMORY EEPROM size`
7. `#pragma AVRPART MEMORY INT_SRAM SIZE size`
8. `#pragma AVRPART MEMORY INT_SRAM START_ADDR address`
9. `#pragma partinclude num`

Description

These directives are used to specify various part-specific properties, and are normally used in the part definition include files (partdef.inc). Normally, there is no reason to use these pragmas directly in user programs.

Preprocessor macros are not allowed in pragmas. Expressions are not allowed for the numeric arguments, must be pure numbers in decimal, hex, octal, or binary format. String arguments must not be quoted. The pragmas specify the following part-specific properties:

1. The part name, e.g., ATmega8.
2. The AVR Core version. This defines the basic instruction set supported. Allowed core versions are currently V0, V0E, V1, V2, and V2E.
3. Colon-separated list of instructions not supported by this part, relative to the core version.
4. Colon-separated list of additional instructions supported by this part, relative to the core version.
5. FLASH program memory size, in bytes.
6. EEPROM memory size, in bytes.
7. SRAM memory size, in bytes.
8. SRAM memory start address. 0x60 for basic AVR parts, 0x100 or more for parts with extended I/O.
9. For AVRASM1 compatibility. Default value: 0. If set to 1, it will cause the `DEVICE` directive to include a file called device.h that is expected to contain the `#pragma`s described above. This enables partdef.inc files designed for AVRASM1 (containing a `DEVICE` directive but no part-describing `#pragma`s) to be used with AVRASM2. This property can also be set using the AVR Assembler Command Line Options command line option.

Examples

Note that the combination of these examples does not describe a real AVR part!

```
1. #pragma AVRPART ADMIN PART_NAME ATmega32
2. #pragma AVRPART CORE CORE_VERSION V2
3. #pragma AVRPART CORE INSTRUCTIONS_NOT_SUPPORTED movw:break:lpm rd,z
4. #pragma AVRPART CORE NEW_INSTRUCTIONS lpm rd,z+
5. #pragma AVRPART MEMORY PROG_FLASH 131072
6. #pragma AVRPART MEMORY EEPROM 4096
7. #pragma AVRPART MEMORY INT_SRAM SIZE 4096
8. #pragma AVRPART MEMORY INT_SRAM START_ADDR 0x60
```

6.12 # (empty directive)

Syntax

#

Description

Unsurprisingly, this directive does exactly nothing. The only reason it exists is that it is required by the ANSI C standard.

6.13 Operators

6.13.1 Stringification (#)

The stringification operators makes a quoted string token of a parameter to a function-type macro.

Example

```
#define MY_IDENT(X) .db #X, '\n', 0
```

When called like this

```
MY_IDENT(FooFirmwareRev1)
```

will expand to

```
.db "FooFirmwareRev1", '\n', 0
```

Notes

1. Stringification can only be used with parameters to function-type macros.
2. The parameter's value will be used literally, i.e. it will not be expanded before stringification.

6.13.2 Concatenation (##)

The concatenation operator concatenates two preprocessor tokens, forming a new token. It is most useful when at least one of the tokens are a parameter to a function-type macro.

Example

```
#define FOOBAR subi
```

1. `#define IMMED(X) X##i`
2. `#define SUBI(X,Y) X ## Y`

When the `IMMED` and `SUBI` macros are called like this:

1. `IMMED(ld) r16,1`
2. `SUBI(FOO,BAR) r16,1`

they will be expanded to

1. `ldi r16,0x1`
2. `subi r16,0x1`

Note:

1. The concatenation operator cannot appear first or last in a macro expansion.
2. When used on a function-type macro argument, the argument will be used literally, i.e., it will not be expanded before concatenation.
3. The token formed by the concatenation will be subject to further expansion. In example 2 above, the parameters `FOO` and `BAR` are first concatenated to `FOOBAR`, then `FOOBAR` is expanded to `subi`.

6.14 Pre-defined Macros

The preprocessor has a number of pre-defined macros. All have names starting and ending with two underscores (__). To avoid conflicts, user-defined macros should not use this naming convention.

Pre-defined macros are either built-in, or they are set by the #pragma, General Purpose directive, as indicated in the table below.

Name	Type	Set by	Description
__AVRASM_VERSION__	Integer	Built-in	Assembler version, encoded as (1000*major + minor)
__CORE_VERSION__	String	#pragma, General Purpose	AVR core version

Name	Type	Set by	Description
__DATE__	String	built-in	Build date. Format: "Jun 28 2004". See AVR Assembler Command Line Options
__TIME__	String	built-in	Build time. Format: "HH:MM:SS". See AVR Assembler Command Line Options
__CENTURY__	Integer	built-in	Build time century (typically 20)
__YEAR__	Integer	built-in	Build time year, less century (0-99)
__MONTH__	Integer	built-in	Build time month (1-12)
__DAY__	Integer	built-in	Build time day (1-31)
__HOUR__	Integer	built-in	Build time hour (0-23)
__MINUTE__	Integer	built-in	Build time minute (0-59)
__SECOND__	Integer	built-in	Build time second (0-59)
__FILE__	String	built-in	Source file name
__LINE__	Integer	built-in	Current line number in source file
__PART_NAME__	String	#pragma, General Purpose	AVR part name
__partname__	Integer	#pragma, General Purpose	partname corresponds to the value of __PART_NAME__. Example: #ifdef __ATmega8__
__CORE_coreversion__	Integer	#pragma, General Purpose	coreversion corresponds to the value of __CORE_VERSION__. Example: #ifdef __CORE_V2__

7. Expressions

The Assembler incorporates constant expressions.

Expressions can consist of operands, operators, and functions. All expressions are 64 bits internally.

7.1 Functions

Functions defined for the assembler.

- LOW(expression) returns the low byte of an expression
- HIGH(expression) returns the second byte of an expression
- BYTE2(expression) is the same function as HIGH
- BYTE3(expression) returns the third byte of an expression
- BYTE4(expression) returns the fourth byte of an expression
- LWRD(expression) returns bits 0-15 of an expression
- HWRD(expression) returns bits 16-31 of an expression
- PAGE(expression) returns bits 16-21 of an expression
- EXP2(expression) returns 2 to the power of expression
- LOG2(expression) returns the integer part of log2(expression)
- INT(expression) Truncates a floating point expression to integer (i.e. discards fractional part)
- FRAC(expression) Extracts fractional part of a floating point expression (i.e. discards integer part).
- Q7(expression) Converts a fractional floating point expression to a form suitable for the `FMUL`/`FMULU`/`FMULSU` instructions (see also http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf). (Sign + 7-bit fraction.)
- Q15(expression) Converts a fractional floating point expression to a form suitable for the `FMUL`/`FMULU`/`FMULSU` instructions (see also http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf). (Sign +15-bit fraction.)
- ABS() Returns the absolute value of a constant expression
- DEFINED(symbol). Returns true if symbol is previously defined using .equ/.set/.def directives. Normally used in conjunction with .if directives (.if defined(foo)), but may be used in any context. It differs from other functions in that parentheses around its argument are not required, and that it only makes sense to use a single symbol as argument.
- STRLEN(string) Returns the length of a string constant, in bytes

7.2 Operands

The following operands can be used:

- User defined labels, which are given the value of the location counter at the place they appear
- User defined variables defined by the SET directive
- User defined constants defined by the EQU directive
- Integer constants: constants can be given in several formats, including:
  - Decimal (default): 10, 255
  - Hexadecimal (two notations): 0x0a, $0a, 0xff, $ff
  - Binary: 0b00001010, 0b11111111
  - Octal (leading zero): 010, 077

- PC - the current value of the Program memory location counter
- Floating point constants

7.3 Operators

The Assembler supports a number of operators, which are described here. The higher the precedence, the higher the priority. Expressions may be enclosed in parentheses, and such expressions are always evaluated before combined with anything outside the parentheses. The associativity of binary operators indicates the evaluation order of chained operators, left associativity meaning they are evaluated left to right, i.e., 2 - 3 - 4 is (2 - 3) - 4, while right associativity would mean 2-3-4 is 2 - (3 - 4). Some operators are not associative, meaning chaining has no meaning.

The following operators are defined:

Symbol	Description
!	Logical not
~	Bitwise Not
-	Unary Minus
*	Multiplication
/	Division
%	Modulo (AVR Assembler 2 only)
+	Addition
-	Subtraction
<<	Shift left
>>	Shift right
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equal
!=	Not equal
&	Bitwise And
^	Bitwise Xor
|	Bitwise Or
&&	Logical And

Symbol	Description
||	Logical Or
?	Conditional operator

Logical Not

Symbol: !

Description: Unary operator, which returns 1 if the expression was zero, and returns 0 if the expression was nonzero.

Precedence: 14

Associativity: None

Example: `ldi r16,!0xf0 ; Load r16 with 0x00`

Bitwise Not

Symbol: ~

Description: Unary operator, which returns the input expression with all bits inverted.

Precedence: 14

Associativity: None

Example: `ldi r16,~0xf0 ; Load r16 with 0x0f`

Unary Minus

Symbol: -

Description: Unary operator, which returns the arithmetic negation of an expression.

Precedence: 14

Associativity: None

Example: `ldi r16,-2 ; Load -2(0xfe) in r16`

Multiplication

Symbol: *

Description: Binary operator, which returns the product of two expressions.

Precedence: 13

Associativity: Left

Example: `ldi r30,label*2 ; Load r30 with label*2`

Division

Symbol: /

Description: Binary operator, which returns the integer quotient of the left expression divided by the right expression.

Precedence: 13

Associativity: Left

Example: `ldi r30,label/2 ; Load r30 with label/2`

Modulo

Symbol: %

Description: Binary operator, which returns the integer remainder of the left expression divided by the right expression.

Precedence: 13

Associativity: Left

Example: `ldi r30,label%2 ; Load r30 with label%2`

Addition

Symbol: +

Description: Binary operator, which returns the sum of two expressions.

Precedence: 12

Associativity: Left

Example: `ldi r30,c1+c2 ; Load r30 with c1+c2`

Subtraction

Symbol: -

Description: Binary operator, which returns the left expression minus the right expression.

Precedence: 12

Associativity: Left

Example: `ldi r17,c1-c2 ;Load r17 with c1-c2`

Shift left

Symbol: <<

Description: Binary operator, which returns the left expression shifted left the number given by the right expression.

Precedence: 11

Associativity: Left

Example: `ldi r17,1<<bitmask ;Load r17 with 1 shifted left bitmask times`

Shift right

Symbol: >>

Description: Binary operator, which returns the left expression shifted right the number given by the right expression.

Precedence: 11

Associativity: Left

Example: `ldi r17,c1>>c2 ;Load r17 with c1 shifted right c2 times`

Less than

Symbol: <

Description: Binary operator, which returns 1 if the signed expression to the left is Less than the signed expression to the right, 0 otherwise.

Precedence: 10

Associativity: None

Example: `ori r18,bitmask*(c1<c2)+1 ;Or r18 with an expression`

Less or equal

Symbol: <=

Description: Binary operator, which returns 1 if the signed expression to the left is Less than or Equal to the signed expression to the right, 0 otherwise.

Precedence: 10

Associativity: None

Example: `ori r18,bitmask*(c1<=c2)+1 ;Or r18 with an expression`

Greater than

Symbol: >

Description: Binary operator, which returns 1 if the signed expression to the left is Greater than the signed expression to the right, 0 otherwise.

Precedence: 10

Associativity: None

Example: `ori r18,bitmask*(c1>c2)+1 ;Or r18 with an expression`

Greater or equal

Symbol: >=

Description: Binary operator, which returns 1 if the signed expression to the left is Greater than or Equal to the signed expression to the right, 0 otherwise.

Precedence: 10

Associativity: None

Example: `ori r18,bitmask*(c1>=c2)+1 ;Or r18 with an expression`

Equal

Symbol: ==

Description: Binary operator, which returns 1 if the signed expression to the left is Equal to the signed expression to the right, 0 otherwise.

Precedence: 9

Associativity: None

Example: `andi r19,bitmask*(c1==c2)+1 ;And r19 with an expression`

Not equal

Symbol: !=

Description: Binary operator, which returns 1 if the signed expression to the left is Not Equal to the signed expression to the right, 0 otherwise.

Precedence: 9

Associativity: None

Example: `.SET flag=(c1!=c2) ;Set flag to 1 or 0`

Bitwise And

Symbol: &

Description: Binary operator, which returns the bitwise And between two expressions.

Precedence: 8

Associativity: Left

Example: `ldi r18,high(c1&c2) ;Load r18 with an expression`

Bitwise Xor

Symbol: ^

Description: Binary operator, which returns the bitwise Exclusive Or between two expressions.

Precedence: 7

Associativity: Left

Example: `ldi r18,low(c1^c2) ;Load r18 with an expression`

Bitwise Or

Symbol: |

Description: Binary operator, which returns the bitwise Or between two expressions.

Precedence: 6

Associativity: Left

Example: `ldi r18,low(c1|c2) ;Load r18 with an expression`

Logical And

Symbol: &&

Description: Binary operator, which returns 1 if the expressions are both nonzero, 0 otherwise.

Precedence: 5

Associativity: Left

Example: `ldi r18,low(c1&&c2) ;Load r18 with an expression`

Logical Or

Symbol: ||

Description: Binary operator, which returns 1 if one or both of the expressions are nonzero, 0 otherwise.

Precedence: 4

Associativity: Left

Example: `ldi r18,low(c1||c2) ;Load r18 with an expression`

Conditional operator

Symbol: ? :

Syntax: condition? expression1 : expression2

Description: Ternary operator, which returns expression1 if condition is true, expression2 otherwise.

Precedence: 3

Associativity: None

Example:

```
ldi r18, a > b? a : b ;Load r18 with the maximum numeric value of a and b.
```

Note: This feature was introduced in AVRASM 2.1 and is not available in 2.0 or earlier versions.

8. AVR Instruction Set

For information about the AVR instruction set, refer to the 8-bit AVR Instruction Set Manual.

9. Revision History

Doc Rev.	Date	Comments
A	06/2017	Complete update of the document. Converted to Microchip format and replaced the Atmel document number 1022.
1022A	01/1998	Initial document release.

The Microchip Web Site

Microchip provides online support via our web site at http://www.microchip.com/. This web site is used as a means to make files and information easily available to customers. Accessible by using your favorite Internet browser, the web site contains the following information:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQ), technical support requests, online discussion groups, Microchip consultant program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

Customer Change Notification Service

Microchip's customer notification service helps keep customers current on Microchip products. Subscribers will receive e-mail notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, access the Microchip web site at http://www.microchip.com/. Under "Support", click on "Customer Change Notification" and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Technical Support

Customers should contact their distributor, representative or Field Application Engineer (FAE) for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in the back of this document.

Technical support is available through the web site at: http://www.microchip.com/support

Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.

- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable.”

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip’s code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Legal Notice

Trademarks

ISBN: 978-1-5224-1828-3

Quality Management System Certified by DNV

ISO/TS 16949

Microchip received ISO/TS-16949:2009 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

Worldwide Sales and Service

AMERICAS	ASIA/PACIFIC	ASIA/PACIFIC	EUROPE
Corporate Office 2355 West Chandler Blvd. Chandler, AZ 85224-6199 Tel: 480-792-7200 Fax: 480-792-7277 Technical Support: http://www.microchip.com/support Web Address: www.microchip.com **Atlanta** Duluth, GA Tel: 678-957-9614 Fax: 678-957-1455 **Austin, TX** Tel: 512-257-3370 **Boston** Westborough, MA Tel: 774-760-0087 Fax: 774-760-0088 **Chicago** Itasca, IL Tel: 630-285-0071 Fax: 630-285-0075 **Dallas** Addison, TX Tel: 972-818-7423 Fax: 972-818-2924 **Detroit** Novi, MI Tel: 248-848-4000 **Houston, TX** Tel: 281-894-5983 **Indianapolis** Noblesville, IN Tel: 317-773-8323 Fax: 317-773-5453 Tel: 317-536-2380 **Los Angeles** Mission Viejo, CA Tel: 949-462-9523 Fax: 949-462-9608 Tel: 951-273-7800 **Raleigh, NC** Tel: 919-844-7510 **New York, NY** Tel: 631-435-6000 **San Jose, CA** Tel: 408-735-9110 Tel: 408-436-4270 **Canada - Toronto** Tel: 905-695-1980 Fax: 905-695-2078	**Asia Pacific Office** Suites 3707-14, 37th Floor Tower 6, The Gateway Harbour City, Kowloon **Hong Kong** Tel: 852-2943-5100 Fax: 852-2401-3431 **Australia - Sydney** Tel: 61-2-9868-6733 Fax: 61-2-9868-6755 **China - Beijing** Tel: 86-10-8569-7000 Fax: 86-10-8528-2104 **China - Chengdu** Tel: 86-28-8665-5511 Fax: 86-28-8665-7889 **China - Chongqing** Tel: 86-23-8980-9588 Fax: 86-23-8980-9500 **China - Dongguan** Tel: 86-769-8702-9880 **China - Guangzhou** Tel: 86-20-8755-8029 **China - Hangzhou** Tel: 86-571-8792-8115 Fax: 86-571-8792-8116 **China - Hong Kong SAR** Tel: 852-2943-5100 Fax: 852-2401-3431 **China - Nanjing** Tel: 86-25-8473-2460 Fax: 86-25-8473-2470 **China - Qingdao** Tel: 86-532-8502-7355 Fax: 86-532-8502-7205 **China - Shanghai** Tel: 86-21-3326-8000 Fax: 86-21-3326-8021 **China - Shenyang** Tel: 86-24-2334-2829 Fax: 86-24-2334-2393 **China - Shenzhen** Tel: 86-755-8864-2200 Fax: 86-755-8203-1760 **China - Wuhan** Tel: 86-27-5980-5300 Fax: 86-27-5980-5118 **China - Xian** Tel: 86-29-8833-7252 Fax: 86-29-8833-7256	**China - Xiamen** Tel: 86-592-2388138 Fax: 86-592-2388130 **China - Zhuhai** Tel: 86-756-3210040 Fax: 86-756-3210049 **India - Bangalore** Tel: 91-80-3090-4444 Fax: 91-80-3090-4123 **India - New Delhi** Tel: 91-11-4160-8631 Fax: 91-11-4160-8632 **India - Pune** Tel: 91-20-3019-1500 **Japan - Osaka** Tel: 81-6-6152-7160 Fax: 81-6-6152-9310 **Japan - Tokyo** Tel: 81-3-6880- 3770 Fax: 81-3-6880-3771 **Korea - Daegu** Tel: 82-53-744-4301 Fax: 82-53-744-4302 **Korea - Seoul** Tel: 82-2-554-7200 Fax: 82-2-558-5932 or 82-2-558-5934 **Malaysia - Kuala Lumpur** Tel: 60-3-6201-9857 Fax: 60-3-6201-9859 **Malaysia - Penang** Tel: 60-4-227-8870 Fax: 60-4-227-4068 **Philippines - Manila** Tel: 63-2-634-9065 Fax: 63-2-634-9069 **Singapore** Tel: 65-6334-8870 Fax: 65-6334-8850 **Taiwan - Hsin Chu** Tel: 886-3-5778-366 Fax: 886-3-5770-955 **Taiwan - Kaohsiung** Tel: 886-7-213-7830 **Taiwan - Taipei** Tel: 886-2-2508-8600 Fax: 886-2-2508-0102 **Thailand - Bangkok** Tel: 66-2-694-1351 Fax: 66-2-694-1350	**Austria - Wels** Tel: 43-7242-2244-39 Fax: 43-7242-2244-393 **Denmark - Copenhagen** Tel: 45-4450-2828 Fax: 45-4485-2829 **Finland - Espoo** Tel: 358-9-4520-820 **France - Paris** Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79 **France - Saint Cloud** Tel: 33-1-30-60-70-00 **Germany - Garching** Tel: 49-8931-9700 **Germany - Haan** Tel: 49-2129-3766400 **Germany - Heilbronn** Tel: 49-7131-67-3636 **Germany - Karlsruhe** Tel: 49-721-625370 **Germany - Munich** Tel: 49-89-627-144-0 Fax: 49-89-627-144-44 **Germany - Rosenheim** Tel: 49-8031-354-560 **Israel - Ra'anana** Tel: 972-9-744-7705 **Italy - Milan** Tel: 39-0331-742611 Fax: 39-0331-466781 **Italy - Padova** Tel: 39-049-7625286 **Netherlands - Drunen** Tel: 31-416-690399 Fax: 31-416-690340 **Norway - Trondheim** Tel: 47-7289-7561 **Poland - Warsaw** Tel: 48-22-3325737 **Romania - Bucharest** Tel: 40-21-407-87-50 **Spain - Madrid** Tel: 34-91-708-08-90 Fax: 34-91-708-08-91 **Sweden - Gothenberg** Tel: 46-31-704-60-40 **Sweden - Stockholm** Tel: 46-8-5090-4654 **UK - Wokingham** Tel: 44-118-921-5800 Fax: 44-118-921-5820