Assembler directives

The Assembler supports a number of directives. The directives are not translated directly into opcodes. Instead, they are used to adjust the location of the program in memory, define macros, initialize memory and so on. An overview of the directives is given in the following table.

Directive	Description
BYTE	Reserve byte to a variable
CSEG	Code Segment
CSEGSIZE	Program memory size
DB	Define constant byte(s)
DEF	Define a symbolic name on a register
DEVICE	Define which device to assemble for
DSEG	Data Segment
DW	Define Constant word(s)
ENDM, ENDMACRO	EndMacro
EQU	Set a symbol equal to an expression
ESEG	EEPROM Segment
EXIT	Exit from file
INCLUDE	Read source from another file
LIST	Turn listfile generation on
LISTMAC	Turn Macro expansion in list file on
MACRO	Begin Macro
NOLIST	Turn listfile generation off
ORG	Set program origin
SET	Set a symbol to an expression

These directives were introduced in AVRASM v1.74:

New Directives	Description
ELSE,ELIF	Conditional assembly
ENDIF	Conditional assembly
ERROR	Outputs an error message
IF,IFDEF,IFNDEF	Conditional assembly
MESSAGE	Outputs a message string

The following directives are only available with AVRASM2:

AVRASM2 Directives	Description
DD	Define Doubleword
DQ	Define Quadword
UNDEF	Undefine register symbol
WARNING	Outputs a warning message
OVERLAP/NOOVERLAP	Set up overlapping section (requires AVRASM2.1 or better)

Note that all directives must be preceded by a period.

BYTE - Reserve bytes to 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 CSEG, DSEG, and ESEG). 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
```

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
```

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
```

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 expressionlist contains more than one expression, the expressions are packed so that two bytes are placed in each program memory word. I*f the expressionlist 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 assemby 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
```

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
```

UNDEF - Undefine a register symbolic name

This directive is only available with AVRASM2.

'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.
```

DEVICE - Define which device to assemble for (deprecated)

Note: *The .DEVICE directive is deprecated and should no longer be used with AVRASM2. It is replaced with a number of #pragma directives, describing the device properties. In AVRASM2, .device is roughly equivalent to* ***`#pragma AVRPART PART_NAME`****, but the device name may be anything and is not limited to the table below. Also, only the device name is set by .device, the other device properties must be set separately. The description and table below is only valid for AVRASM 1.x, ant the table is not updated with new devices.*

The DEVICE directive allows the user to tell the Assembler which device the code is to be executed on. Using this directive, a warning is issued if an instruction not supported by the specified device occurs. If the Code Segment or EEPROM Segment are larger than supplied by the device, a warning message is given. If the directive is not used, it is assumed that all instructions are supported and that there are no restrictions on Program and EEPROM memory.

Syntax:

```
.DEVICE <device code>
```

Table: Device codes for devices supported by AVRASM 1.x:

Classic	Tiny	Mega	Other
AT90S120	ATtiny11	ATmega8	AT94K
AT90S4433	ATtiny2313	ATmega48	AT86RF401
AT90S2313	ATtiny12	ATmega16	AT90CAN128
AT90S2323	ATtiny13	ATmega161	
AT90S2333	ATtiny22	ATmega162	
AT90S4414	ATtiny26	ATmega163	
AT90S4434	ATtiny25	ATmega169	
AT90S8515	ATtiny45	ATmega32	
AT90S8534	ATtiny85	ATmega323	
AT90S8535		ATmega103	
AT90S2343		ATmega104	
		ATmega8515	
		ATmega8535	
		ATmega64	
		ATmega128	
		ATmega165	
		ATmega2560	
		ATmega2561	

Example:

```
.DEVICE AT90S1200  ; Use the AT90S1200
```

```
.CSEG
        push r30   ; This statement will generate a warning
                   ; since the specified device does not
                   ; have this instruction
```

Note: There has been a change of names that took effect 14.06.2001. The following devices are affected:

Old name	New name
ATmega104	ATmega128
ATmega32	ATmega323
ATmega164	ATmega16

In order NOT to break old projects, both old and new device directives are allowed for the parts that are affected.

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 only consist of BYTE directives (and labels). 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
```

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

The DW directive reserves memory resources in the program memory or the EEPROM memory. 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 memory location.

Syntax:

```
LABEL: .DW expressionlist
```

Example:

```
.CSEG
```

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

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

DD - Define constant doubleword(s) in program memory and EEPROM
DQ - Define constant quadword(s) in program memory and EEPROM

These directives are only available with AVRASM2.

These directives are very similar to the .DW directive, except they are used to define 32-bit (doubleword) and 64-bit (quadword) respectively. The data layout in memory is strictly little-endian.

Syntax:
```
LABEL: .DD expressionlist
LABEL: .DQ expressionlist
```

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

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

ELIF,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>
...
.ENDIF

Example:
.IFDEF DEBUG
.MESSAGE "Debugging.."
.ELSE
.MESSAGE "Release.."
.ENDIF

ENDIF - conditional assembly

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

ENDM, 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
```

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
```

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
```

WARNING - Outputs a warning message string

The .WARNING directive outputs a warning string, but unlike the .ERROR directive 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
```

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 only consist of DB and DW directives (and labels). 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
```

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
```

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 itself contain INCLUDE directives.

Syntax:

```
.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
```

IF,IFDEF,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 <expression> is evaluated different from 0. Valid till the corresponding ELSE or ENDIF directive.

Up to 5 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

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
```

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
```

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
```

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
```

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
```

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
```

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
```

OVERLAP/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/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 overlap 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).

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
```

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.