Usage

Command line option overview

The assembler accepts the following options:

```
------------------------------------------------------------------
Usage: ca65 [options] file
Short options:
  -D name[=value]       Define a symbol
  -I dir                Set an include directory search path
  -U                    Mark unresolved symbols as import
  -V                    Print the assembler version
  -W n                  Set warning level n
  -g                    Add debug info to object file
  -h                    Help (this text)
  -i                    Ignore case of symbols
  -l                    Create a listing if assembly was ok
  -o name               Name the output file
  -s                    Enable smart mode
  -t sys                Set the target system
  -v                    Increase verbosity

Long options:
  --auto-import         Mark unresolved symbols as import
  --debug-info          Add debug info to object file
  --feature name        Set an emulation feature
  --help                Help (this text)
  --ignore-case         Ignore case of symbols
  --include-dir dir     Set an include directory search path
  --listing             Create a listing if assembly was ok
  --pagelength n        Set the page length for the listing
  --verbose             Increase verbosity
  --version             Print the assembler version
```

Command line options in detail

Here is a description of all the command line options:

--feature name

Enable an emulation feature. This is identical as using .FEATURE in the source with two exceptions: Feature names must be lower case, and each feature must be specified by using an extra --feature option, comma separated lists are not allowed. See the discussion of the .FEATURE command for a list of emulation features.

-g, --debug-info

When this option (or the equivalent control command .DEBUGINFO) is used, the assembler will add a section to the object file that contains all symbols (including local ones) together with the symbol values and source file positions. The linker will put these additional symbols into the VICE label file, so even local symbols can be seen in the VICE monitor.

-h, --help

Print the short option summary shown above.

-i, --ignore-case

This option makes the assembler case insensitive on identifiers and labels. This option will override the default, but may itself be overriden by the .CASE control command.

-l, --listing

Generate an assembler listing. The listing file will always have the name of the main input file with the extension replaced by ".lst". This may change in future versions.

-o name

The default output name is the name of the input file with the extension replaced by ".o". If you don't like that, you may give another name with the -o option. The output file will be placed in the same directory as the source file, or, if -o is given, the full path in this name is used.

--pagelength n

sets the length of a listing page in lines. See the .PAGELENGTH directive for more information.

-v, --verbose

Increase the assembler verbosity. Usually only needed for debugging purposes. You may use this option more than one time for even more verbose output.

-D

This option allows you to define symbols on the command line. Without a value, the symbol is defined with the value zero. When giving a value, you may use the '$' prefix for hexadecimal symbols. Please note that for some operating systems, '$' has a special meaning, so you may have to quote the expression.

-I dir, --include-dir dir

Name a directory which is searched for include files. The option may be used more than once to specify more than one directory to search. The current directory is always searched first before considering any additional directores.

-U, --auto-import

Mark symbols that are not defined in the sources as imported symbols. This should be used with care since it delays error messages about typos and such until the linker is run. The compiler uses the equivalent of this switch (.AUTOIMPORT) to enable auto imported symbols for the runtime library. However, the compiler is supposed to generate code that runs through the assembler without problems, something which is not always true for assembler programmers.

-V, --version

Print the version number of the assembler. If you send any suggestions or bugfixes, please include the version number.

-Wn

Set the warning level for the assembler. Using -W2 the assembler will even warn about such things like unused imported symbols. The default warning level is 1, and it would probably be silly to set it to something lower.

Expressions

Expression evaluation

All expressions are evaluated with (at least) 32 bit precision. An expression may contain constant values and any combination of internal and external symbols. Expressions that cannot be evaluated at assembly time are stored inside the object file for evaluation by the linker. Expressions referencing imported symbols must always be evaluated by the linker.

Size of an expression result

Sometimes, the assembler must know about the size of the value that is the result of an expression. This is usually the case, if a decision has to be made, to generate a zero page or an absolute memory references. In this case, the assembler has to make some assumptions about the result of an expression:

- If the result of an expression is constant, the actual value is checked to see if it's a byte sized expression or not.
- If the expression is explicitly casted to a byte sized expression by one of the '>', '<' or '^' operators, it is a byte expression.

- If this is not the case, and the expression contains a symbol, explicitly declared as zero page symbol (by one of the .importzp or .exportzp instructions), then the whole expression is assumed to be byte sized.
- If the expression contains symbols that are not defined, and these symbols are local symbols, the enclosing scopes are searched for a symbol with the same name. If one exists and this symbol is defined, it's attributes are used to determine the result size.
- In all other cases the expression is assumed to be word sized.

Note: If the assembler is not able to evaluate the expression at assembly time, the linker will evaluate it and check for range errors as soon as the result is known.

Boolean expressions

In the context of a boolean expression, any non zero value is evaluated as true, any other value to false. The result of a boolean expression is 1 if it's true, and zero if it's false. There are boolean operators with extrem low precedence with version 2.x (where x > 0). The .AND and .OR operators are shortcut operators. That is, if the result of the expression is already known, after evaluating the left hand side, the right hand side is not evaluated.

Constant expressions

Sometimes an expression must evaluate to a constant without looking at any further input. One such example is the .IF command that decides if parts of the code are assembled or not. An expression used in the .IF command cannot reference a symbol defined later, because the decision about the .IF must be made at the point when it is read. If the expression used in such a context contains only constant numerical values, there is no problem. When unresolvable symbols are involved it may get harder for the assembler to determine if the expression is actually constant, and it is even possible to create expressions that aren't recognized as constant. Simplifying the expressions will often help.

In cases where the result of the expression is not needed immediately, the assembler will delay evaluation until all input is read, at which point all symbols are known. So using arbitrary complex constant expressions is no problem in most cases.

Available operators

Available operators sorted by precedence:

Op	Description	Precedence
	Builtin string functions	0
	Builtin pseudo variables	1
	Builtin pseudo functions	1
+	Unary plus	1

-	Unary minus	1
~	Unary bitwise not	1
.BITNOT	Unary bitwise not	1
<	Low byte operator	1
>	High byte operator	1
^	Bank byte operator	1
*	Multiplication	2
/	Division	2
.MOD	Modulo operation	2
&	Bitwise and	2
.BITAND	Bitwise and	2
^	Bitwise xor	2
.BITXOR	Bitwise xor	2
<<	Shift left operator	2
.SHL	Shift left operator	2
>>	Shift right operator	
.SHR	Shift right operator	2
+	Binary plus	3
-	Binary minus	3
|	Binary or	3
.BITOR	Binary or	3
=	Compare operation (equal)	4
<>	Compare operation (not equal)	4
<	Compare operation (less)	4
>	Compare operation (greater)	4
<=	Compare operation (less or equal)	4
>=	Compare operation (greater or equal)	4
&&	Boolean and	5
.AND	Boolean and	5
.XOR	Boolean xor	5
||	Boolean or	6
.OR	Boolean or	6
!	Boolean not	7
.NOT	Boolean not	7

To force a specific order of evaluation, braces may be used as usual.

Symbols and labels

The assembler allows you to use symbols instead of naked values to make the source more readable. There are a lot of different ways to define and use symbols and labels, giving a lot of flexibility.

Numeric constants

Numeric constants are defined using the equal sign or the label assignment operator. After doing

```
two = 2
```

may use the symbol "two" in every place where a number is expected, and it is evaluated to the value 2 in this context. The label assignment operator causes the same, but causes the symbol to be marked as a label, which may cause a different handling in the debugger:

```
io := $d000
```

The right side can of course be an expression:

```
four = two * two
```

Standard labels

A label is defined by writing the name of the label at the start of the line (before any instruction mnemonic, macro or pseudo directive), followed by a colon. This will declare a symbol with the given name and the value of the current program counter.

Local labels and symbols

Using the .PROC directive, it is possible to create regions of code where the names of labels and symbols are local to this region. They are not known outside of this region and cannot be accessed from there. Such regions may be nested like PROCEDUREs in Pascal.

See the description of the .PROC directive for more information.

Cheap local labels

Cheap local labels are defined like standard labels, but the name of the label must begin with a special symbol (usually '@', but this can be changed by the .LOCALCHAR directive).
Cheap local labels are visible only between two non cheap labels. As soon as a standard symbol is encountered (this may also be a local symbol if inside a region defined with the .PROC directive), the cheap local symbol goes out of scope.

You may use cheap local labels as an easy way to reuse common label names like "Loop". Here is an example:

```
Clear:  lda     #$00            ; Global label
        ldy     #$20
@Loop:  sta     Mem,y           ; Local label
        dey
        bne     @Loop           ; Ok
        rts
Sub:    ...                     ; New global label
        bne     @Loop           ; ERROR: Unknown identifier!
```

Unnamed labels

If you really want to write messy code, there are also unnamed labels. These labels do not have a name (you guessed that already, didn't you?). A colon is used to mark the absence of the name.

Unnamed labels may be accessed by using the colon plus several minus or plus characters as a label designator. Using the '-' characters will create a back reference (use the n'th label backwards), using '+' will create a forward reference (use the n'th label in forward direction). An example will help to understand this:

```
:       lda     (ptr1),y        ; #1
        cmp     (ptr2),y
        bne     :+              ; -> #2
        tax
        beq     :+++            ; -> #4
        iny
        bne     :-              ; -> #1
        inc     ptr1+1
        inc     ptr2+1
        bne     :-              ; -> #1

:       bcs     :+              ; #2 -> #3
        ldx     #$FF
        rts

:       ldx     #$01            ; #3
:       rts                     ; #4
```

As you can see from the example, unnamed labels will make even short sections of code hard to understand, because you have to count labels to find branch targets (this is the reason why I for my part do prefer the "cheap" local labels). Nevertheless, unnamed labels are convenient in some situations, so it's your decision.

Using macros to define labels and constants

While there are drawbacks with this approach, it may be handy in some situations. Using .DEFINE, it is possible to define symbols or constants that may be used elsewhere. Since the macro facility works on a very low level, there is no scoping. On the other side, you may also define string constants this way (this is not possible with the other symbol types).

Example:

```
.DEFINE two     2
.DEFINE version "SOS V2.3"

four = two * two        ; Ok
.byte   version         ; Ok

.PROC                   ; Start local scope
two = 3                 ; Will give "2 = 3" - invalid!
.ENDPROC
```

Symbols and .DEBUGINFO

If .DEBUGINFO is enabled (or -g is given on the command line), global, local and cheap local labels are written to the object file and will be available in the symbol file via the linker. Unnamed labels are not written to the object file, because they don't have a name which would allow to access them.

Scopes

Global scope

All (non cheap local) symbols that are declared outside of any nested scopes are in global scope.

A special scope: cheap locals

A special scope is the scope for cheap local symbols. It lasts from one non local symbol to the next one, without any provisions made by the programmer. All other scopes differ in usage but use the same concept internally.

Generic nested scopes

A nested scoped for generic use is started with .SCOPE and closed with .ENDSCOPE. The scope can have a name, in which case it is accessible from the outside by using explicit scopes. If the scope does not have a name, all symbols created within the scope are local to the scope, and aren't accessible from the outside.
A nested scope can access symbols from the local or from enclosing scopes by name without using explicit scope names. In some cases there may be ambiguities, for

example if there is a reference to a local symbol that is not yet defined, but a symbol with the same name exists in outer scopes:

```
        .scope  outer
                foo     = 2
                .scope  inner
                        lda     #foo
                        foo     = 3
                .endscope
        .endscope
```

In the example above, the lda instruction will load the value 3 into the accumulator, because foo is redefined in the scope. However:

```
        .scope  outer
                foo     = $1234
                .scope  inner
                        lda     foo,x
                        foo     = $12
                .endscope
        .endscope
```

Here, lda will still load from $12,x, but since it is unknown to the assembler that foo is a zeropage symbol when translating the instruction, absolute mode is used instead. In fact, the assembler will not use absolute mode by default, but it will search through the enclosing scopes for a symbol with the given name. If one is found, the address size of this symbol is used. This may lead to errors:

```
        .scope  outer
                foo     = $12
                .scope  inner
                        lda     foo,x
                        foo     = $1234
                .endscope
        .endscope
```

In this case, when the assembler sees the symbol foo in the lda instruction, it will search for an already defined symbol foo. It will find foo in scope outer, and a close look reveals that it is a zeropage symbol. So the assembler will use zeropage addressing mode. If foo is redefined later in scope inner, the assembler tries to change the address in the lda instruction already translated, but since the new value needs absolute addressing mode, this fails, and an error message "Range error" is output.

Of course the most simple solution for the problem is to move the definition of foo in scope inner upwards, so it preceeds its use. There may be rare cases when this cannot be done. In these cases, you can use one of the address size override operators:

```
        .scope  outer
                foo     = $12
                .scope  inner
                        lda     a:foo,x
                        foo     = $1234
                .endscope
        .endscope
```

This will cause the lda instruction to be translated using absolute addressing mode, which means changing the symbol reference later does not cause any errors.

Nested procedures

A nested procedure is created by use of .PROC. It differs from a .SCOPE in that it must have a name, and a it will introduce a symbol with this name in the enclosing scope. So

```
        .proc   foo
                ...
        .endscope
```

is actually the same as

```
        foo:
        .scope  foo
                ...
        .endscope
```

This is the reason why a procedure must have a name. If you want a scope without a name, use .SCOPE.

Note: As you can see from the example above, scopes and symbols live in different namespaces. There can be a symbol named foo and a scope named foo without any conflicts (but see the section titled "Scope search order").

Structs, unions and enums

Structs, unions and enums are explained in a separate section, I do only cover them here, because if they are declared with a name, they open a nested scope, similar to .SCOPE. However, when no name is specified, the behaviour is different: In this case, no new scope will be opened, symbols declared within a struct, union, or enum declaration will then be added to the enclosing scope instead.

Explicit scope specification

Accessing symbols from other scopes is possible by using an explicit scope specification, provided that the scope where the symbol lives in has a name. The namespace token (::) is used to access other scopes:

```
        .scope  foo
        bar:    .word   0
        .endscope

                ...
                lda     foo::bar        ; Access foo in scope bar
```

The only way to deny access to a scope from the outside is to declare a scope without a name (using the .SCOPE command).

A special syntax is used to specify the global scope: If a symbol or scope is preceeded by the namespace token, the global scope is searched:

```
        bar     = 3

        .scope  foo
                bar     = 2
                lda     #::bar  ; Access the global bar (which is 3)
        .endscope
```

Scope search order

The assembler searches for a scope in a similar way as for a symbol. First, it looks in the current scope, and then it walks up the enclosing scopes until the scope is found.

However, one important thing to note when using explicit scope syntax is, that a symbol may be accessed before it is defined, but a scope may **not** be used without a preceeding definition. This means that in the following example:

```
        .scope  foo
                bar     = 3
        .endscope

        .scope  outer
                lda     #foo::bar  ; Will load 3, not 2!
                .scope  foo
                        bar     = 2
                .endscope
        .endscope
```

the reference to the scope foo will use the global scope, and not the local one, because the local one is not visible at the point where it is referenced.

Things get more complex if a complete chain of scopes is specified:

```
        .scope  foo
                .scope  outer
                        .scope  inner
                                bar = 1
                        .endscope
                .endscope
                .scope  another
                        .scope  nested
                                lda     #outer::inner::bar      ; 1
                        .endscope
                .endscope
        .endscope

        .scope  outer
                .scope  inner
                        bar = 2
                .endscope
        .endscope
```

When outer::inner::bar is referenced in the lda instruction, the assembler will first search in the local scope for a scope named outer. Since none is found, the enclosing scope (another) is checked. There is still no scope named outer, so scope foo is checked, and finally scope outer is found. Within this scope, inner is searched, and in this scope, the assembler looks for a symbol named bar.

Please note that once the anchor scope is found, all following scopes (inner in this case) are expected to be found exactly in this scope. The assembler will search the scope tree only for the first scope (if it is not anchored in the root scope). Starting from there on, there is no flexibility, so if the scope named outer found by the assembler does not contain a scope named inner, this would be an error, even if such a pair does exist (one level up in global scope).

Ambiguities that may be introduced by this search algorithm may be removed by anchoring the scope specification in the global scope. In the example above, if you want to access the "other" symbol bar, you would have to write:

```
        .scope  foo
```

```
                .scope  outer
                        .scope  inner
                                bar = 1
                        .endscope
                .endscope
                .scope  another
                        .scope  nested
                                lda     #::outer::inner::bar    ; 2
                        .endscope
                .endscope
        .endscope

        .scope  outer
                .scope  inner
                        bar = 2
                .endscope
        .endscope
```

Pseudo functions

Pseudo functions expect their arguments in parenthesis, and they have a result, either a string or an expression.

.BANKBYTE

The function returns the bank byte (that is, bits 16-23) of its argument. It works identical to the '^' operator.

See: .HIBYTE, .LOBYTE

.BLANK

Builtin function. The function evaluates its argument in braces and yields "false" if the argument is non blank (there is an argument), and "true" if there is no argument. As an example, the .IFBLANK statement may be replaced by

```
        .if     .blank(arg)
```

.CONCAT

Builtin string function. The function allows to concatenate a list of string constants separated by commas. The result is a string constant that is the concatentation of all arguments. This function is most useful in macros and when used together with the .STRING builtin function. The function may be used in any case where a string constant is expected.

Example:

```
        .include        .concat ("myheader", ".", "inc")
```

This is the same as the command

```
        .include        "myheader.inc"
```

.CONST

Builtin function. The function evaluates its argument in braces and yields "true" if the argument is a constant expression (that is, an expression that yields a constant value at assembly time) and "false" otherwise. As an example, the .IFCONST statement may be replaced by

```
        .if     .const(a + 3)
```

.HIBYTE

The function returns the high byte (that is, bits 8-15) of its argument. It works identical to the '>' operator.

See: .LOBYTE, .BANKBYTE

.HIWORD

The function returns the high word (that is, bits 16-31) of its argument.

See: .LOWORD

.LEFT

Builtin function. Extracts the left part of a given token list.
Syntax:

```
        .LEFT (<int expr>, <token list>)
```

The first integer expression gives the number of tokens to extract from the token list. The second argument is the token list itself.
Example:
To check in a macro if the given argument has a '#' as first token (immidiate addressing mode), use something like this:

```
        .macro  ldax    arg
                ...
                .if (.match (.left (1, arg), #))

                ; ldax called with immidiate operand
                ...

                .endif
                ...
        .endmacro
```

See also the .MID and .RIGHT builtin functions.

.LOBYTE

The function returns the low byte (that is, bits 0-7) of its argument. It works identical to the '<' operator.

See: .HIBYTE, .BANKBYTE

.LOWORD

The function returns the low word (that is, bits 0-15) of its argument.

See: .HIWORD

.MATCH

Builtin function. Matches two token lists against each other. This is most useful within macros, since macros are not stored as strings, but as lists of tokens. The syntax is

```
.MATCH(<token list #1>, <token list #2>)
```

Both token list may contain arbitrary tokens with the exception of the terminator token (comma resp. right parenthesis) and

- end-of-line
- end-of-file

Often a macro parameter is used for any of the token lists.

Please note that the function does only compare tokens, not token attributes. So any number is equal to any other number, regardless of the actual value. The same is true for strings. If you need to compare tokens *and* token attributes, use the .XMATCH function.

Example:

Assume the macro ASR, that will shift right the accumulator by one, while honoring the sign bit. The builtin processor instructions will allow an optional "A" for accu addressing for instructions like ROL and ROR. We will use the .MATCH function to check for this and print and error for invalid calls.

```
        .macro  asr     arg

                .if (.not .blank(arg)) .and (.not .match (arg, a))
                .error "Syntax error"
                .endif

                cmp     #$80            ; Bit 7 into carry
                lsr     a               ; Shift carry into bit 7
```

```
.endmacro
```

The macro will only accept no arguments, or one argument that must be the reserved keyword "A".

See: .XMATCH

.MID

Builtin function. Takes a starting index, a count and a token list as arguments. Will return part of the token list.

Syntax:

```
.MID (<int expr>, <int expr>, <token list>)
```

The first integer expression gives the starting token in the list (the first token has index 0). The second integer expression gives the number of tokens to extract from the token list. The third argument is the token list itself.

Example:

To check in a macro if the given argument has a '#' as first token (immidiate addressing mode), use something like this:

```
.macro  ldax    arg
        ...
        .if (.match (.mid (0, 1, arg), #))

        ; ldax called with immidiate operand
        ...

        .endif
        ...
.endmacro
```

See also the .LEFT and .RIGHT builtin functions.

.REF, .REFERENCED

Builtin function. The function expects an identifier as argument in braces. The argument is evaluated, and the function yields "true" if the identifier is a symbol that has already been referenced somewhere in the source file up to the current position. Otherwise the function yields false. As an example, the .IFREF statement may be replaced by

```
        .if     .referenced(a)
```

See: .DEFINED

.RIGHT

Builtin function. Extracts the right part of a given token list.

Syntax:

```
        .RIGHT (<int expr>, <token list>)
```

The first integer expression gives the number of tokens to extract from the token list. The second argument is the token list itself.

See also the .LEFT and .MID builtin functions.

.SIZEOF

.SIZEOF is a pseudo function that returns the size of its argument. The argument can be a struct/union, a struct member, a procedure, or a label. In case of a procedure or label, its size is defined by the amount of data placed in the segment where the label is relative to. If a line of code switches segments (for example in a macro) data placed in other segments does not count for the size.

Please note that a symbol or scope must exist, before it is used together with .SIZEOF (this may get relaxed later, but will always be true for scopes). A scope has preference over a symbol with the same name, so if the last part of a name represents both, a scope and a symbol, the scope is choosen over the symbol.

After the following code:

```
        .struct Point                   ; Struct size = 4
                xcoord  .word
                xcoord  .word
        .endstruct

        P:      .tag    Point           ; Declare a point
        @P:     .tag    Point           ; Declare another point

        .code
        .proc   Code
                nop
                .proc   Inner
                        nop
                .endproc
                nop
        .endproc
```

```
        .proc   Data
        .data                           ; Segment switch!!!
                .res    4
        .endproc
```

.sizeof(Point)

will have the value 4, because this is the size of struct Point.

.sizeof(Point::xcoord)

will have the value 2, because this is the size of the member xcoord in struct Point.

.sizeof(P)

will have the value 4, this is the size of the data declared on the same source line as the label P, which is in the same segment that P is relative to.

.sizeof(@P)

will have the value 4, see above. The example demonstrates that .SIZEOF does also work for cheap local symbols.

.sizeof(Code)

will have the value 3, since this is amount of data emitted into the code segment, the segment that was active when Code was entered. Note that this value includes the amount of data emitted in child scopes (in this case Code::Inner).

.sizeof(Code::Inner)

will have the value 1 as expected.

.sizeof(Data)

will have the value 0. Data is emitted within the scope Data, but since the segment is switched after entry, this data is emitted into another segment.

.STRAT

Builtin function. The function accepts a string and an index as arguments and returns the value of the character at the given position as an integer value. The index is zero based.

Example:

```
        .macro  M       Arg
                ; Check if the argument string starts with '#'
                .if (.strat (Arg, 0) = '#')
                ...
                .endif
        .endmacro
```

.STRING

Builtin function. The function accepts an argument in braces and converts this argument into a string constant. The argument may be an identifier, or a constant numeric value.

Since you can use a string in the first place, the use of the function may not be obvious. However, it is useful in macros, or more complex setups.

Example:

```
        ; Emulate other assemblers:
        .macro  section name
                .segment        .string(name)
         .endmacro
```

.STRLEN

Builtin function. The function accepts a string argument in braces and eveluates to the length of the string.

Example:

The following macro encodes a string as a pascal style string with a leading length byte.

```
        .macro  PString Arg
                .byte   .strlen(Arg), Arg
        .endmacro
```

.TCOUNT

Builtin function. The function accepts a token list in braces. The function result is the number of tokens given as argument.

Example:

The ldax macro accepts the '#' token to denote immidiate addressing (as with the normal 6502 instructions). To translate it into two separate 8 bit load instructions, the '#' token has to get stripped from the argument:

```
        .macro  ldax    arg
                .if (.match (.mid (0, 1, arg), #))
                ; ldax called with immidiate operand
                lda     #<(.right (.tcount (arg)-1, arg))
                ldx     #>(.right (.tcount (arg)-1, arg))
                .else
                ...
                .endif
        .endmacro
```

.XMATCH

Builtin function. Matches two token lists against each other. This is most useful within macros, since macros are not stored as strings, but as lists of tokens. The syntax is

```
        .XMATCH(<token list #1>, <token list #2>)
```

Both token list may contain arbitrary tokens with the exception of the terminator token (comma resp. right parenthesis) and

- end-of-line
- end-of-file

Often a macro parameter is used for any of the token lists.
The function compares tokens *and* token values. If you need a function that just compares the type of tokens, have a look at the .MATCH function.
See: .MATCH

Control commands

.ADDR

Define word sized data. In 6502 mode, this is an alias for .WORD and may be used for better readability if the data words are address values. In 65816 mode, the address is forced to be 16 bit wide to fit into the current segment. See also .FARADDR.
The command must be followed by a sequence of (not necessarily constant) expressions.
Example:

```
        .addr   $0D00, $AF13, _Clear
```

See: .FARADDR, .WORD

.ALIGN

Align data to a given boundary. The command expects a constant integer argument that must be a power of two, plus an optional second argument in byte range. If there is a second argument, it is used as fill value, otherwise the value defined in the linker configuration file is used (the default for this value is zero).
Since alignment depends on the base address of the module, you must give the same (or a greater) alignment for the segment when linking. The linker will give you a warning, if you don't do that.
Example:

```
        .align  256
```

.ASCIIZ

Define a string with a trailing zero.
Example:

```
        Msg:    .asciiz "Hello world"
```

This will put the string "Hello world" followed by a binary zero into the current segment. There may be more strings separated by commas, but the binary zero is only appended once (after the last one).

.ASSERT

Add an assertion. The command is followed by an expression, an action specifier and a message that is output in case the assertion fails. The action specifier may be one of warning or error. The assertion is passed to the linker and will be evaluated when segment placement has been done.

Example:

```
        .assert         * = $8000, error, "Code not at $8000"
```

The example assertion will check that the current location is at $8000, when the output file is written, and abort with an error if this is not the case. More complex expressions are possible. The action specifier warning outputs a warning, while the error specifier outputs an error message. In the latter case, generation if the output file is suppressed.

.AUTOIMPORT

Is followed by a plus or a minus character. When switched on (using a +), undefined symbols are automatically marked as import instead of giving errors. When switched off (which is the default so this does not make much sense), this does not happen and an error message is displayed. The state of the autoimport flag is evaluated when the complete source was translated, before outputing actual code, so it is *not* possible to switch this feature on or off for separate sections of code. The last setting is used for all symbols.

You should probably not use this switch because it delays error messages about undefined symbols until the link stage. The cc65 compiler (which is supposed to produce correct assembler code in all circumstances, something which is not true for most assembler programmers) will insert this command to avoid importing each and every routine from the runtime library.

Example:

```
        .autoimport     +       ; Switch on auto import
```

.BSS

Switch to the BSS segment. The name of the BSS segment is always "BSS", so this is a shortcut for

```
        .segment  "BSS"
```

See also the .SEGMENT command.

.BYT, .BYTE

Define byte sized data. Must be followed by a sequence of (byte ranged) expressions or strings.

Example:

```
        .byte   "Hello "
        .byt    "world", $0D, $00
```

.CASE

Switch on or off case sensitivity on identifiers. The default is off (that is, identifiers are case sensitive), but may be changed by the -i switch on the command line. The command must be followed by a '+' or '-' character to switch the option on or off respectively.

Example:

```
        .case   -               ; Identifiers are not case sensitive
```

.CHARMAP

Apply a custom mapping for characters. The command is followed by two numbers in the range 1..255. The first one is the index of the source character, the second one is the mapping. The mapping applies to all character and string constants when they generate output, and overrides a mapping table specified with the -t command line switch.

Example:

```
        .charmap        $41, $61        ; Map 'A' to 'a'
```

.CODE

Switch to the CODE segment. The name of the CODE segment is always "CODE", so this is a shortcut for

```
        .segment  "CODE"
```

See also the .SEGMENT command.

.CONDES

Export a symbol and mark it in a special way. The linker is able to build tables of all such symbols. This may be used to automatically create a list of functions needed to initialize linked library modules.

Note: The linker has a feature to build a table of marked routines, but it is your code that must call these routines, so just declaring a symbol with .CONDES does nothing by itself.

All symbols are exported as an absolute (16 bit) symbol. You don't need to use an additional .EXPORT statement, this is implied by .CONDES.

.CONDES is followed by the type, which may be constructor, destructor or a numeric value between 0 and 6 (where 0 is the same as specifiying constructor and 1 is equal to specifying destructor). The .CONSTRUCTOR and .DESTRUCTOR commands are actually shortcuts for .CONDES with a type of constructor resp. destructor.

After the type, an optional priority may be specified. Higher numeric values mean higher priority. If no priority is given, the default priority of 7 is used. Be careful when assigning priorities to your own module constructors so they won't interfere with the ones in the cc65 library.

Example:

```
        .condes         ModuleInit, constructor
        .condes         ModInit, 0, 16
```

See the .CONSTRUCTOR and .DESTRUCTOR commands and the separate section Module constructors/destructors explaining the feature in more detail.

.CONSTRUCTOR

Export a symbol and mark it as a module constructor. This may be used together with the linker to build a table of constructor subroutines that are called by the startup code.

Note: The linker has a feature to build a table of marked routines, but it is your code that must call these routines, so just declaring a symbol as constructor does nothing by itself.

A constructor is always exported as an absolute (16 bit) symbol. You don't need to use an additional .export statement, this is implied by .constructor. It may have an optional priority that is separated by a comma. Higher numeric values mean a higher priority. If no priority is given, the default priority of 7 is used. Be careful when assigning priorities to your own module constructors so they won't interfere with the ones in the cc65 library.

Example:

```
        .constructor    ModuleInit
        .constructor    ModInit, 16
```

See the .CONDES and .DESTRUCTOR commands and the separate section Module constructors/destructors explaining the feature in more detail.

.DATA

Switch to the DATA segment. The name of the DATA segment is always "DATA", so this is a shortcut for

```
        .segment  "DATA"
```

See also the .SEGMENT command.

.DBYT

Define word sized data with the hi and lo bytes swapped (use .WORD to create word sized data in native 65XX format). Must be followed by a sequence of (word ranged) expressions.

Example:

```
        .dbyt   $1234, $4512
```

This will emit the bytes

```
        $12 $34 $45 $12
```

into the current segment in that order.

.DEBUGINFO

Switch on or off debug info generation. The default is off (that is, the object file will not contain debug infos), but may be changed by the -g switch on the command line. The command must be followed by a '+' or '-' character to switch the option on or off respectively.

Example:

```
        .debuginfo      +       ; Generate debug info
```

.DEFINE

Start a define style macro definition. The command is followed by an identifier (the macro name) and optionally by a list of formal arguments in braces. See section Macros.

.DEF, .DEFINED

Builtin function. The function expects an identifier as argument in braces. The argument is evaluated, and the function yields "true" if the identifier is a symbol that is already defined somewhere in the source file up to the current position. Otherwise the function yields false. As an example, the .IFDEF statement may be replaced by

```
        .if     .defined(a)
```

.DESTRUCTOR

Export a symbol and mark it as a module destructor. This may be used together with the linker to build a table of destructor subroutines that are called by the startup code.

Note: The linker has a feature to build a table of marked routines, but it is your code that must call these routines, so just declaring a symbol as constructor does nothing by itself.

A destructor is always exported as an absolute (16 bit) symbol. You don't need to use an additional .export statement, this is implied by .destructor. It may have an optional priority that is separated by a comma. Higher numerical values mean a higher priority. If no priority is given, the default priority of 7 is used. Be careful when assigning priorities to your own module destructors so they won't interfere with the ones in the cc65 library.
Example:

```
        .destructor     ModuleDone
        .destructor     ModDone, 16
```

See the .CONDES and .CONSTRUCTOR commands and the separate section Module constructors/destructors explaining the feature in more detail.

.DWORD
Define dword sized data (4 bytes) Must be followed by a sequence of expressions.
Example:

```
        .dword  $12344512, $12FA489
```

.ELSE
Conditional assembly: Reverse the current condition.

.ELSEIF

Conditional assembly: Reverse current condition and test a new one.

.END

Forced end of assembly. Assembly stops at this point, even if the command is read from an include file.

.ENDENUM

End a .ENUM declaration.

.ENDIF

Conditional assembly: Close a .IF... or .ELSE branch.

.ENDMAC, .ENDMACRO

End of macro definition (see section Macros).

.ENDPROC

End of local lexical level (see .PROC).

.ENDREP, .ENDREPEAT

End a .REPEAT block.

.ENDSCOPE

End of local lexical level (see .SCOPE).

.ENDSTRUCT

Ends a struct definition. See the .STRUCT command and the separate section named "Structs and unions".

.ENUM

Start an enumeration. This directive is very similar to the C enum keyword. If a name is given, a new scope is created for the enumeration, otherwise the enumeration members are placed in the enclosing scope.

In the enumeration body, symbols are declared. The first symbol has a value of zero, and each following symbol will get the value of the preceeding plus one. This behaviour may be overriden by an explicit assignment. Two symbols may have the same value.

Example:

```
        .enum   errorcodes
                no_error
                file_error
                parse_error
        .endenum
```

Above example will create a new scope named errorcodes with three symbols in it that get the values 0, 1 and 2 respectively. Another way to write this would have been:

```
        .scope  errorcodes
                no_error        = 0
                file_error      = 1
                parse_error     = 2
        .endscope
```

Please note that explicit scoping must be used to access the identifiers:

```
        .word   errorcodes::no_error
```

A more complex example:

```
        .enum
                EUNKNOWN        = -1
                EOK
                EFILE
                EBUSY
                EAGAIN
                EWOULDBLOCK     = EAGAIN
        .endenum
```

In this example, the enumeration does not have a name, which means that the members will be visible in the enclosing scope and can be used in this scope without explicit scoping. The first member (EUNKNOWN) has the value -1. The value for the following members is incremented by one, so EOK would be zero and so on. EWOULDBLOCK is an alias for EGAIN, so it has an override for the value using an already defined symbol.

.ERROR

Force an assembly error. The assembler will output an error message preceeded by "User error" and will *not* produce an object file.

This command may be used to check for initial conditions that must be set before assembling a source file.

Example:

```
        .if     foo = 1
        ...
        .elseif bar = 1
        ...
        .else
        .error  "Must define foo or bar!"
        .endif
```

See also the .WARNING and .OUT directives.

.EXITMAC, .EXITMACRO

Abort a macro expansion immidiately. This command is often useful in recursive macros. See separate section Macros.

.EXPORT

Make symbols accessible from other modules. Must be followed by a comma separated list of symbols to export, with each one optionally followed by an address specification. The default is to export the symbol with the address size it actually has. The assembler will issue a warning, if the symbol is exported with an address size smaller than the actual address size.

Example:

```
.export foo
.export bar: far
```

See: .EXPORTZP

.EXPORTZP

Make symbols accessible from other modules. Must be followed by a comma separated list of symbols to export. The exported symbols are explicitly marked as zero page symols.

Example:

```
.exportzp  foo, bar
```

See: .EXPORT

.FARADDR

Define far (24 bit) address data. The command must be followed by a sequence of (not necessarily constant) expressions.

Example:

```
.faraddr        DrawCircle, DrawRectangle, DrawHexagon
```

See: .ADDR

.FEATURE

This directive may be used to enable one or more compatibility features of the assembler. While the use of .FEATURE should be avoided when possible, it may be useful when porting sources written for other assemblers. There is no way to switch a feature off, once you have enabled it, so using

will enable the feature until end of assembly is reached.

The following features are available:

dollar_is_pc

The dollar sign may be used as an alias for the star (`*'), which gives the value of the current PC in expressions. Note: Assignment to the pseudo variable is not allowed.

labels_without_colons

Allow labels without a trailing colon. These labels are only accepted, if they start at the beginning of a line (no leading white space).

loose_string_term

Accept single quotes as well as double quotes as terminators for string constants.

loose_char_term

Accept single quotes as well as double quotes as terminators for char constants.

at_in_identifiers

Accept the at character (`@') as a valid character in identifiers. The at character is not allowed to start an identifier, even with this feature enabled.

dollar_in_identifiers

Accept the dollar sign (`$') as a valid character in identifiers. The at character is not allowed to start an identifier, even with this feature enabled.

leading_dot_in_identifiers

Accept the dot (`.') as the first character of an identifier. This may be used for example to create macro names that start with a dot emulating control directives of other assemblers. Note however, that none of the reserved

keywords built into the assembler, that starts with a dot, may be overridden. When using this feature, you may also get into trouble if later versions of the assembler define new keywords starting with a dot.

pc_assignment

Allow assignments to the PC symbol (`*' or `$' if dollar_is_pc is enabled). Such an assignment is handled identical to the .ORG command (which is usually not needed, so just removing the lines with the assignments may also be an option when porting code written for older assemblers).

It is also possible to specify features on the command line using the --feature command line option. This is useful when translating sources written for older assemblers, when you don't want to change the source code.

As an example, to translate sources written for Andre Fachats xa65 assembler, the features

labels_without_colons, pc_assignment, loose_char_term

may be helpful. They do not make ca65 completely compatible, so you may not be able to translate the sources without changes, even when enabling these features. However, I have found several sources that translate without problems when enabling these features on the command line.

.FILEOPT, .FOPT

Insert an option string into the object file. There are two forms of this command, one specifies the option by a keyword, the second specifies it as a number. Since usage of the second one needs knowledge of the internal encoding, its use is not recommended and I will only describe the first form here.

The command is followed by one of the keywords

author
comment
compiler

a comma and a string. The option is written into the object file together with the string value. This is currently unidirectional and there is no way to actually use these options once they are in the object file.

Examples:

```
.fileopt        comment, "Code stolen from my brother"
.fileopt        compiler, "BASIC 2.0"
.fopt           author, "J. R. User"
```

.FORCEIMPORT

Import an absolute symbol from another module. The command is followed by a comma separated list of symbols to import. The command is similar to .IMPORT, but the import reference is always written to the generated object file, even if the symbol is never referenced (.IMPORT will not generate import references for unused symbols).

Example:

.forceimport needthisone, needthistoo

See: .IMPORT

.GLOBAL

Declare symbols as global. Must be followed by a comma separated list of symbols to declare. Symbols from the list, that are defined somewhere in the source, are exported, all others are imported. Additional .IMPORT or .EXPORT commands for the same symbol are allowed.

Example:

```
.global foo, bar
```

.GLOBALZP

Declare symbols as global. Must be followed by a comma separated list of symbols to declare. Symbols from the list, that are defined somewhere in the source, are exported, all others are imported. Additional .IMPORTZP or .EXPORTZP commands for the same symbol are allowed. The symbols in the list are explicitly marked as zero page symols.

Example:

```
.globalzp foo, bar
```

.IF

Conditional assembly: Evalute an expression and switch assembler output on or off depending on the expression. The expression must be a constant expression, that is, all operands must be defined.

A expression value of zero evaluates to FALSE, any other value evaluates to TRUE.

.IFBLANK

Conditional assembly: Check if there are any remaining tokens in this line, and evaluate to FALSE if this is the case, and to TRUE otherwise. If the condition is not true, further lines are not assembled until an .ESLE, .ELSEIF or .ENDIF directive.

This command is often used to check if a macro parameter was given. Since an empty macro parameter will evaluate to nothing, the condition will evaluate to FALSE if an empty parameter was given.

Example:

```
        .macro      arg1, arg2
        .ifblank    arg2
                    lda     #arg1
        .else
                    lda     #arg2
        .endif
        .endmacro
```

See also: .BLANK

.IFCONST

Conditional assembly: Evaluate an expression and switch assembler output on or off depending on the constness of the expression.

A const expression evaluates to to TRUE, a non const expression (one containing an imported or currently undefined symbol) evaluates to FALSE.

See also: .CONST

.IFDEF

Conditional assembly: Check if a symbol is defined. Must be followed by a symbol name. The condition is true if the the given symbol is already defined, and false otherwise.

See also: .DEFINED

.IFNBLANK

Conditional assembly: Check if there are any remaining tokens in this line, and evaluate to TRUE if this is the case, and to FALSE otherwise. If the condition is not true, further lines are not assembled until an .ELSE, .ELSEIF or .ENDIF directive.

This command is often used to check if a macro parameter was given. Since an empty macro parameter will evaluate to nothing, the condition will evaluate to FALSE if an empty parameter was given.

Example:

```
        .macro      arg1, arg2
                    lda     #arg1
        .ifnblank   arg2
                    lda     #arg2
        .endif
        .endmacro
```

See also: .BLANK

.IFNDEF

Conditional assembly: Check if a symbol is defined. Must be followed by a symbol name. The condition is true if the the given symbol is not defined, and false otherwise.

See also: .DEFINED

.IFNREF

Conditional assembly: Check if a symbol is referenced. Must be followed by a symbol name. The condition is true if if the the given symbol was not referenced before, and false otherwise.

See also: .REFERENCED

.IFREF

Conditional assembly: Check if a symbol is referenced. Must be followed by a symbol name. The condition is true if if the the given symbol was referenced before, and false otherwise.

This command may be used to build subroutine libraries in include files (you may use separate object modules for this purpose too).

Example:

```
        .ifref  ToHex                   ; If someone used this subroutine
        ToHex:  tay                     ; Define subroutine
                lda     HexTab,y
                rts
        .endif
```

See also: .REFERENCED

.IMPORT

Import a symbol from another module. The command is followed by a comma separated list of symbols to import, with each one optionally followed by an address specification.

Example:

```
        .import foo
        .import bar: zeropage
```

See: .IMPORTZP

.IMPORTZP

Import a symbol from another module. The command is followed by a comma separated list of symbols to import. The symbols are explicitly imported as zero page symbols (that is, symbols with values in byte range).

Example:

```
        .importzp       foo, bar
```

See: .IMPORT

.INCBIN

Include a file as binary data. The command expects a string argument that is the name of a file to include literally in the current segment. In addition to that, a start offset and a size value may be specified, separated by commas. If no size is specified, all of the file from the start offset to end-of-file is used. If no start position is specified either, zero is assume (which means that the whole file is inserted).

Example:

```
        ; Include whole file
        .incbin         "sprites.dat"

        ; Include file starting at offset 256
        .incbin         "music.dat", $100

        ; Read 100 bytes starting at offset 200
        .incbin         "graphics.dat", 200, 100
```

.INCLUDE

Include another file. Include files may be nested up to a depth of 16.

Example:

```
        .include        "subs.inc"
```

.LINECONT

Switch on or off line continuations using the backslash character before a newline. The option is off by default. Note: Line continuations do not work in a comment. A backslash at the end of a comment is treated as part of the comment and does not trigger line continuation. The command must be followed by a '+' or '-' character to switch the option on or off respectively.

Example:

```
        .linecont       +               ; Allow line continuations

        lda     \
                #$20                    ; This is legal now
```

.LIST

Enable output to the listing. The command must be followed by a boolean switch ("on", "off", "+" or "-") and will enable or disable listing output. The option has no effect if the listing is not enabled by the command line switch -l. If -l is used, an internal counter is set to 1. Lines are output to the listing file, if the counter is greater than zero, and suppressed if the counter is zero. Each use of .LIST will increment or decrement the counter.

Example:

```
        .list   on              ; Enable listing output
```

.LISTBYTES

Set, how many bytes are shown in the listing for one source line. The default is 12, so the listing will show only the first 12 bytes for any source line that generates more than 12 bytes of code or data. The directive needs an argument, which is either "unlimited", or an integer constant in the range 4..255.

Examples:

```
        .listbytes      unlimited       ; List all bytes
        .listbytes      12              ; List the first 12 bytes
        .incbin         "data.bin"      ; Include large binary file
```

.LOCAL

This command may only be used inside a macro definition. It declares a list of identifiers as local to the macro expansion.

A problem when using macros are labels: Since they don't change their name, you get a "duplicate symbol" error if the macro is expanded the second time. Labels declared with .LOCAL have their name mapped to an internal unique name (___ABCD__) with each macro invocation.

Some other assemblers start a new lexical block inside a macro expansion. This has some drawbacks however, since that will not allow *any* symbol to be visible outside a macro, a feature that is sometimes useful. The .LOCAL command is in my eyes a better way to address the problem.

You get an error when using .LOCAL outside a macro.

.LOCALCHAR

Defines the character that start "cheap" local labels. You may use one of '@' and '?' as start character. The default is '@'.

Cheap local labels are labels that are visible only between two non cheap labels. This way you can reuse identifiers like "loop" without using explicit lexical nesting.

Example:

```
        .localchar      '?'

Clear:  lda     #$00            ; Global label
?Loop:  sta     Mem,y           ; Local label
        dey
        bne     ?Loop           ; Ok
        rts
Sub:    ...                     ; New global label
```

```
                bne     ?Loop           ; ERROR: Unknown identifier!
```

.MACPACK

Insert a predefined macro package. The command is followed by an identifier specifying the macro package to insert. Available macro packages are:

generic	Defines generic macros like add and sub.
longbranch	Defines conditional long jump macros.
cbm	Defines the scrcode macro

Including a macro package twice, or including a macro package that redefines already existing macros will lead to an error.

Example:

```
        .macpack        longbranch      ; Include macro package

                cmp     #$20            ; Set condition codes
                jne     Label           ; Jump long on condition
```

Macro packages are explained in more detail in section Macro packages.

.MAC, .MACRO

Start a classic macro definition. The command is followed by an identifier (the macro name) and optionally by a comma separated list of identifiers that are macro parameters.

See section Macros.

.ORG

Start a section of absolute code. The command is followed by a constant expression that gives the new PC counter location for which the code is assembled. Use .RELOC to switch back to relocatable code.

Please note that you *do not need* this command in most cases. Placing code at a specific address is the job of the linker, not the assembler, so there is usually no reason to assemble code to a specific address.

You may not switch segments while inside a section of absolute code.

Example:

```
        .org    $7FF            ; Emit code starting at $7FF
```

.OUT

Output a string to the console without producing an error. This command is similiar to .ERROR, however, it does not force an assembler error that prevents the creation of an object file.

Example:

```
        .out    "This code was written by the codebuster(tm)"
```

See also the .WARNING and .ERROR directives.

.PAGELEN, .PAGELENGTH

Set the page length for the listing. Must be followed by an integer constant. The value may be "unlimited", or in the range 32 to 127. The statement has no effect if no listing is generated. The default value is -1 (unlimited) but may be overridden by the --pagelength command line option. Beware: Since ca65 is a one pass assembler, the listing is generated after assembly is complete, you cannot use multiple line lengths with one source. Instead, the value set with the last .PAGELENGTH is used.

Examples:

```
        .pagelength     66              ; Use 66 lines per listing page

        .pagelength     unlimited       ; Unlimited page length
```

.POPSEG

Pop the last pushed segment from the stack, and set it.

This command will switch back to the segment that was last pushed onto the segment stack using the .PUSHSEG command, and remove this entry from the stack.

The assembler will print an error message if the segment stack is empty when this command is issued.

See: .PUSHSEG

.PROC

Start a nested lexical level with the given name and adds a symbol with this name to the enclosing scope. All new symbols from now on are in the local lexical level and are accessible from outside only via explicit scope specification. Symbols defined outside this local level may be accessed as long as their names are not used for new symbols inside the level. Symbols names in other lexical levels do not clash, so you may use the same names for identifiers. The lexical level ends when the .ENDPROC command is read. Lexical levels may be nested up to a depth of 16 (this is an artificial limit to protect against errors in the source).

Note: Macro names are always in the global level and in a separate name space. There is no special reason for this, it's just that I've never had any need for local macro definitions.

Example:

```
.proc   Clear           ; Define Clear subroutine, start new level
        lda     #$00
L1:     sta     Mem,y   ; L1 is local and does not cause a
                        ; duplicate symbol error if used in other
                        ; places
        dey
        bne     L1      ; Reference local symbol
        rts
.endproc                ; Leave lexical level
```

See: .ENDPROC and .SCOPE

.PUSHSEG

Push the currently active segment onto a stack. The entries on the stack include the name of the segment and the segment type. The stack has a size of 16 entries.

.PUSHSEG allows together with .POPSEG to switch to another segment and to restore the old segment later, without even knowing the name and type of the current segment.

The assembler will print an error message if the segment stack is already full, when this command is issued.

See: .POPSEG

.REPEAT

Repeat all commands between .REPEAT and .ENDREPEAT constant number of times. The command is followed by a constant expression that tells how many times the commands in the body should get repeated. Optionally, a comma and an identifier may be specified. If this identifier is found in the body of the repeat statement, it is replaced by the current repeat count (starting with zero for the first time the body is repeated).

.REPEAT statements may be nested. If you use the same repeat count identifier for a nested .REPEAT statement, the one from the inner level will be used, not the one from the outer level.

Example:

The following macro will emit a string that is "encrypted" in that all characters of the string are XORed by the value $55.

```
        .macro  Crypt   Arg
                .repeat .strlen(Arg), I
                .byte   .strat(Arg, I) .xor $55
                .endrep
        .endmacro
```

See: .ENDREPEAT

.RELOC

Switch back to relocatable mode. See the .ORG command.

.RES

Reserve storage. The command is followed by one or two constant expressions. The first one is mandatory and defines, how many bytes of storage should be defined. The second, optional expression must by a constant byte value that will be used as value of the data. If there is no fill value given, the linker will use the value defined in the linker configuration file (default: zero).

Example:

```
        ; Reserve 12 bytes of memory with value $AA
        .res    12, $AA
```

.RODATA

Switch to the RODATA segment. The name of the RODATA segment is always "RODATA", so this is a shortcut for

```
.segment  "RODATA"
```

The RODATA segment is a segment that is used by the compiler for readonly data like string constants.

See also the .SEGMENT command.

.SCOPE

Start a nested lexical level with the given name. All new symbols from now on are in the local lexical level and are accessible from outside only via explicit scope specification. Symbols defined outside this local level may be accessed as long as their names are not used for new symbols inside the level. Symbols names in other lexical levels do not clash, so you may use the same names for identifiers. The lexical level ends when the .ENDSCOPE command is read. Lexical levels may be nested up to a depth of 16 (this is an artificial limit to protect against errors in the source).

Note: Macro names are always in the global level and in a separate name space. There is no special reason for this, it's just that I've never had any need for local macro definitions.

Example:

```
.scope  Error                   ; Start new scope named Error
        None = 0                ; No error
        File = 1                ; File error
        Parse = 2               ; Parse error
.endproc                        ; Close lexical level

        ...
        lda #Error::File        ; Use symbol from scope Error
```

See: .ENDSCOPE and .PROC

.SEGMENT

Switch to another segment. Code and data is always emitted into a segment, that is, a named section of data. The default segment is "CODE". There may be up to 254 different segments per object file (and up to 65534 per executable). There are shortcut commands for the most common segments ("CODE", "DATA" and "BSS").

The command is followed by a string containing the segment name (there are some constraints for the name - as a rule of thumb use only those segment names that would also be valid identifiers). There may also be an optional attribute separated by a colon. Valid attributes are "zeropage" and "absolute".

When specifying a segment for the first time, "absolute" is the default. For all other uses, the attribute specified the first time is the default.

"absolute" means that this is a segment with absolute addressing. That is, the segment will reside somewhere in core memory outside the zero page. "zeropage" means the opposite: The segment will be placed in the zero page and direct (short) addressing is possible for data in this segment.

Beware: Only labels in a segment with the zeropage attribute are marked as reachable by short addressing. The `*' (PC counter) operator will work as in other segments and will create absolute variable values.

Example:

```
        .segment "ROM2"             ; Switch to ROM2 segment
        .segment "ZP2": zeropage    ; New direct segment
        .segment "ZP2"              ; Ok, will use last attribute
        .segment "ZP2": absolute    ; Error, redecl mismatch
```

See: .BSS, .CODE, .DATA and .RODATA

.STRUCT

Starts a struct definition. Structs are covered in a separate section named "Structs and unions".

See: .ENDSTRUCT

.TAG

Allocate space for a struct or union.
Example:

```
        .struct Point
                xcoord  .word
                ycoord  .word
        .endstruct

        .bss
                .tag    Point           ; Allocate 4 bytes
```

.WARNING

Force an assembly warning. The assembler will output a warning message preceeded by "User warning". This warning will always be output, even if other warnings are disabled with the -W0 command line option.

This command may be used to output possible problems when assembling the source file.
Example:

```
        .macro  jne     target
                .local L1
                .ifndef target
                .warning "Forward jump in jne, cannot optimize!"
                beq     L1
                jmp     target
        L1:
                .else
                ...
                .endif
        .endmacro
```

See also the .ERROR and .OUT directives.

.WORD

Define word sized data. Must be followed by a sequence of (word ranged, but not necessarily constant) expressions.

Example:

```
        .word   $0D00, $AF13, _Clear
```

.ZEROPAGE

Switch to the ZEROPAGE segment and mark it as direct (zeropage) segment. The name of the ZEROPAGE segment is always "ZEROPAGE", so this is a shortcut for

```
        .segment  "ZEROPAGE": zeropage
```

Because of the "zeropage" attribute, labels declared in this segment are addressed using direct addressing mode if possible. You *must* instruct the linker to place this segment somewhere in the address range 0..$FF otherwise you will get errors.

See: .SEGMENT

Macros

Introduction

Macros may be thought of as "parametrized super instructions". Macros are sequences of tokens that have a name. If that name is used in the source file, the macro is "expanded", that is, it is replaced by the tokens that were specified when the macro was defined.

Macros without parameters

In it's simplest form, a macro does not have parameters. Here's an example:

```
        .macro  asr             ; Arithmetic shift right
                cmp     #$80    ; Put bit 7 into carry
                ror             ; Rotate right with carry
        .endmacro
```

The macro above consists of two real instructions, that are inserted into the code, whenever the macro is expanded. Macro expansion is simply done by using the name, like this:

```
        lda     $2010
        asr
        sta     $2010
```

Parametrized macros

When using macro parameters, macros can be even more useful:

```
        .macro  inc16   addr
                clc
                lda     addr
                adc     #$01
                sta     addr
                lda     addr+1
                adc     #$00
                sta     addr+1
        .endmacro
```

When calling the macro, you may give a parameter, and each occurence of the name "addr" in the macro definition will be replaced by the given parameter. So

```
        inc16   $1000
```

will be expanded to

```
                clc
                lda     $1000
                adc     #$01
                sta     $1000
                lda     $1000+1
                adc     #$00
                sta     $1000+1
```

A macro may have more than one parameter, in this case, the parameters are separated by commas. You are free to give less parameters than the macro actually takes in the definition. You may also leave intermediate parameters empty. Empty parameters are replaced by empty space (that is, they are removed when the macro is exanded). If you have a look at our macro definition above, you will see, that replacing the "addr" parameter by nothing will lead to wrong code in most lines. To help you, writing macros with a variable parameter list, there are some control commands:

.IFBLANK tests the rest of the line and returns true, if there are any tokens on the remainder of the line. Since empty parameters are replaced by nothing, this may be used to test if a given parameter is empty. .IFNBLANK tests the opposite.

Look at this example:

```
.macro  ldaxy   a, x, y
.ifnblank       a
        lda     #a
.endif
.ifnblank       x
        ldx     #x
.endif
.ifnblank       y
        ldy     #y
.endif
.endmacro
```

This macro may be called as follows:

```
ldaxy   1, 2, 3         ; Load all three registers

ldaxy   1, , 3          ; Load only a and y

ldaxy   , , 3           ; Load y only
```

There's another helper command for determining, which macro parameters are valid: .PARAMCOUNT This command is replaced by the parameter count given, *including* intermediate empty macro parameters:

```
ldaxy   1               ; .PARAMCOUNT = 1
ldaxy   1,,3            ; .PARAMCOUNT = 3
ldaxy   1,2             ; .PARAMCOUNT = 2
ldaxy   1,              ; .PARAMCOUNT = 2
ldaxy   1,2,3           ; .PARAMCOUNT = 3
```

Detecting parameter types

Sometimes it is nice to write a macro that acts differently depending on the type of the argument supplied. An example would be a macro that loads a 16 bit value from either an immediate operand, or from memory. The .MATCH and .XMATCH functions will allow you to do exactly this:

```
        .macro  ldax    arg
                .if (.match (.left (1, arg), #))
                    ; immediate mode
                    lda     #<(.right (.tcount (arg)-1, arg))
                    ldx     #>(.right (.tcount (arg)-1, arg))
                .else
                    ; assume absolute or zero page
                    lda     arg
                    ldx     1+(arg)
                .endif
        .endmacro
```

Using the .MATCH function, the macro is able to check if its argument begins with a hash mark. If so, two immediate loads are emitted, Otherwise a load from an absolute zero page memory location is assumed. So this macro can be used as

```
        foo:    .word   $5678
        ...
                ldax    #$1234          ; X=$12, A=$34
        ...
                ldax    foo             ; X=$56, A=$78
```

Recursive macros

Macros may be used recursively:

```
        .macro  push    r1, r2, r3
                lda     r1
                pha
        .if     .paramcount > 1
                push    r2, r3
        .endif
        .endmacro
```

There's also a special macro to help writing recursive macros: .EXITMACRO This command will stop macro expansion immidiately:

```
.macro  push    r1, r2, r3, r4, r5, r6, r7
.ifblank        r1
        ; First parameter is empty
        .exitmacro
.else
        lda     r1
        pha
.endif
        push    r2, r3, r4, r5, r6, r7
.endmacro
```

When expanding this macro, the expansion will push all given parameters until an empty one is encountered. The macro may be called like this:

```
push    $20, $21, $32           ; Push 3 ZP locations
push    $21                     ; Push one ZP location
```

Local symbols inside macros

Now, with recursive macros, .IFBLANK and .PARAMCOUNT, what else do you need? Have a look at the inc16 macro above. Here is it again:

```
.macro  inc16   addr
        clc
        lda     addr
        adc     #$01
        sta     addr
        lda     addr+1
        adc     #$00
        sta     addr+1
.endmacro
```

If you have a closer look at the code, you will notice, that it could be written more efficiently, like this:

```
.macro  inc16   addr
        inc     addr
        bne     Skip
        inc     addr+1
Skip:
.endmacro
```

But imagine what happens, if you use this macro twice? Since the label "Skip" has the same name both times, you get a "duplicate symbol" error. Without a way to circumvent this problem, macros are not as useful, as they could be. One solution is, to start a new lexical block inside the macro:

```
.macro  inc16   addr
.proc
        inc     addr
        bne     Skip
        inc     addr+1
Skip:
.endproc
.endmacro
```

Now the label is local to the block and not visible outside. However, sometimes you want a label inside the macro to be visible outside. To make that possible, there's a new command that's only usable inside a macro definition: .LOCAL. .LOCAL declares one or more symbols as local to the macro expansion. The names of local variables are replaced by a unique name in each separate macro expansion. So we could also solve the problem above by using .LOCAL:

```
.macro  inc16   addr
        .local  Skip            ; Make Skip a local symbol
        clc
        lda     addr
        adc     #$01
        sta     addr
        bcc     Skip
        inc     addr+1
Skip:                           ; Not visible outside
.endmacro
```

C style macros

Starting with version 2.5 of the assembler, there is a second macro type available: C style macros using the .DEFINE directive. These macros are similar to the classic macro type described above, but behaviour is sometimes different:

- Macros defined with .DEFINE may not span more than a line. You may use line continuation (see .LINECONT) to spread the definition over more than one line for increased readability, but the macro itself may not contain an end-of-line token.
- Macros defined with .DEFINE share the name space with classic macros, but they are detected and replaced at the scanner level. While classic macros may be used in every place, where a mnemonic or other directive is allowed, .DEFINE style macros are allowed anywhere in a line. So they are more versatile in some situations.

- .DEFINE style macros may take parameters. While classic macros may have empty parameters, this is not true for .DEFINE style macros. For this macro type, the number of actual parameters must match exactly the number of formal parameters. To make this possible, formal parameters are enclosed in braces when defining the macro. If there are no parameters, the empty braces may be omitted.
- Since .DEFINE style macros may not contain end-of-line tokens, there are things that cannot be done. They may not contain several processor instructions for example. So, while some things may be done with both macro types, each type has special usages. The types complement each other.

Let's look at a few examples to make the advantages and disadvantages clear.

To emulate assemblers that use "EQU" instead of "=" you may use the following .DEFINE:

```
.define EQU     =

foo     EQU     $1234           ; This is accepted now
```

You may use the directive to define string constants used elsewhere:

```
; Define the version number
.define VERSION         "12.3a"

; ... and use it
.asciiz VERSION
```

Macros with parameters may also be useful:

```
.define DEBUG(message)  .out    message

DEBUG   "Assembling include file #3"
```

Note that, while formal parameters have to be placed in braces, this is not true for the actual parameters. Beware: Since the assembler cannot detect the end of one parameter, only the first token is used. If you don't like that, use classic macros instead:

```
.macro  message
        .out    message
.endmacro
```

(This is an example where a problem can be solved with both macro types).

Characters in macros

When using the -t option, characters are translated into the target character set of the specific machine. However, this happens as late as possible. This means that strings are translated if they are part of a .BYTE or .ASCIIZ command. Characters are translated as soon as they are used as part of an expression.

This behaviour is very intuitive outside of macros but may be confusing when doing more complex macros. If you compare characters against numeric values, be sure to take the translation into account.

Macro packages

Using the .MACPACK directive, predefined macro packages may be included with just one command. Available macro packages are:

.MACPACK generic

This macro package defines macros that are useful in almost any program. Currently, two macros are defined:

```
.macro  add     Arg
        clc
        adc     Arg
.endmacro

.macro  sub     Arg
        sec
        sbc     Arg
.endmacro
```

.MACPACK longbranch

This macro package defines long conditional jumps. They are named like the short counterpart but with the 'b' replaced by a 'j'. Here is a sample definition for the "jeq" macro, the other macros are built using the same scheme:

```
.macro  jeq     Target
        .if     .def(Target) .and ((*+2)-(Target) <= 127)
        beq     Target
```

```
                .else
                bne     *+5
                jmp     Target
                .endif
        .endmacro
```

All macros expand to a short branch, if the label is already defined (back jump) and is reachable with a short jump. Otherwise the macro expands to a conditional branch with the branch condition inverted, followed by an absolute jump to the actual branch target.

The package defines the following macros:

```
jeq, jne, jmi, jpl, jcs, jcc, jvs, jvc
```

.MACPACK cbm

The cbm macro package will define a macro named scrcode. It takes a string as argument and places this string into memory translated into screen codes.

Structs and unions

Structs and unions are special forms of scopes. They are to some degree comparable to their C counterparts. Both have a list of members. Each member allocates storage and may optionally have a name, which, in case of a struct, is the offset from the beginning and, in case of a union, is always zero.

Here is an example for a very simple struct with two members and a total size of 4 bytes:

```
.struct Point
        xcoord  .word
        ycoord  .word
.endstruct
```

A union shares the total space between all its members, its size is the same as that of the largest member.

A struct or union must not necessarily have a name. If it is anonymous, no local scope is opened, the identifiers used to name the members are placed into the current scope instead.

A struct may contain unnamed members and definitions of local structs. The storage allocators may contain a multiplier, as in the example below:

```
.struct Circle
        .struct Point
                .word   2       ; Allocate two words
        .endstruct
        Radius  .word
.endstruct
```

Using the .TAG keyword, it is possible to embedd already defined structs or unions in structs:

```
.struct Point
        xcoord  .word
        ycoord  .word
.endstruct

.struct Circle
        Origin  .tag    Point
        Radius  .byte
.endstruct
```

Space for a struct or union may be allocated using the .TAG directive.

```
C:      .tag    Circle
```

Currently, members are just offsets from the start of the struct or union. To access a field of a struct, the member offset has to be added to the address of the struct itself:

```
lda     C+Circle::Radius        ; Load circle radius into A
```

This may change in a future version of the assembler.

Module constructors/destructors

Note: This section applies mostly to C programs, so the explanation below uses examples from the C libraries. However, the feature may also be useful for assembler programs.

Module overview

Using the .CONSTRUCTOR and .DESTRUCTOR keywords it it possible to export functions in a special way. The linker is able to generate tables with all functions of a specific type. Such a table will *only* include symbols from object files that are linked into a specific executable. This may be used to add initialization and cleanup code for library modules.

The C heap functions are an example where module initialization code is used. All heap functions (malloc, free, ...) work with a few variables that contain the start and the end of the heap, pointers to the free list and so on. Since the end of the heap depends on the size and start of the stack, it must be initialized at runtime. However, initializing these variables for programs that do not use the heap are a waste of time and memory.

So the central module defines a function that contains initialization code and exports this function using the .CONSTRUCTOR statement. If (and only if) this module is added to an executable by the linker, the initialization function will be placed into the table of constructors by the linker. The C startup code will call all constructors before main and all destructors after main, so without any further work, the heap initialization code is called once the module is linked in.

While it would be possible to add explicit calls to initialization functions in the startup code, the new approach has several advantages:

1. If a module is not included, the initialization code is not linked in and not called. So you don't pay for things you don't need.
2. Adding another library that needs initialization does not mean that the startup code has to be changed. Before we had module constructors and destructors, the startup code for all systems had to be adjusted to call the new initialization code.
3. The feature saves memory: Each additional initialization function needs just two bytes in the table (a pointer to the function).

Calling order

Both, constructors and destructors are sorted in increasing priority order by the linker when using one of the builtin linker configurations, so the functions with lower priorities come first and are followed by those with higher priorities. The C library runtime subroutine that walks over the constructor and destructor tables calls the functions starting from the top of the table - which means that functions with a high priority are called first.
So when using the C runtime, both constructors and destructors are called with high priority functions first, followed by low priority functions.

Pitfalls

When creating and using module constructors and destructors, please take care of the following:

- The linker will only generate function tables, it will not generate code to call these functions. If you're using the feature in some other than the existing C environments, you have to write code to call all functions in a linker generated table yourself. See the condes module in the C runtime for an example on how to do this.
- The linker will only add addresses of functions that are in modules linked to the executable. This means that you have to be careful where to place the condes functions. If initialization is needed for a group of functions, be sure to place the initialization function into a module that is linked in regardless of which function is called by the user.
- The linker will generate the tables only when requested to do so by the FEATURE CONDES statement in the linker config file. Each table has to be requested separately.
- Constructors and destructors may have priorities. These priorities determine the order of the functions in the table. If your intialization or cleanup code does depend on other initialization or cleanup code, you have to choose the priority for the functions accordingly.
- Besides the .CONSTRUCTOR and .DESTRUCTOR statements, there is also a more generic command: .CONDES. This allows to specify an additional type. Predefined types are 0 (constructor) and 1 (destructor). The linker generates a separate table for each type on request.