ca65 Users Guide

Ullrich von Bassewitz,
Greg King

2016-06-11

---

ca65 is a powerful macro assembler for the 6502, 65C02, and 65816 CPUs. It is used as a companion assembler for the cc65 crosscompiler, but it may also be used as a standalone product.

---

1. Overview

- 1.1 Design criteria

2. Usage

- 2.1 Command line option overview
- 2.2 Command line options in detail

3. Search paths

4. Input format

- 4.1 Assembler syntax
- 4.2 65816 mode
- 4.3 6502X mode
- 4.4 4510 mode
- 4.5 sweet16 mode
- 4.6 Number format
- 4.7 Conditional assembly

5. Expressions

- 5.1 Expression evaluation
- 5.2 Size of an expression result
- 5.3 Boolean expressions
- 5.4 Constant expressions
- 5.5 Available operators

6. Symbols and labels

- 6.1 Numeric constants
- 6.2 Numeric variables
- 6.3 Standard labels
- 6.4 Local labels and symbols
- 6.5 Cheap local labels
- 6.6 Unnamed labels
- 6.7 Using macros to define labels and constants
- 6.8 Symbols and `.DEBUGINFO`

7. Scopes

- 7.1 Global scope
- 7.2 Cheap locals
- 7.3 Generic nested scopes
- 7.4 Nested procedures
- 7.5 Structs, unions and enums
- 7.6 Explicit scope specification
- 7.7 Scope search order

8. Address sizes and memory models

- 8.1 Address sizes
- 8.2 Address sizes of segments
- 8.3 Address sizes of symbols
- 8.4 Memory models

9. Pseudo variables

- 9.1 *
- 9.2 .ASIZE
- 9.3 .CPU
- 9.4 .ISIZE
- 9.5 .PARAMCOUNT
- 9.6 .TIME
- 9.7 .VERSION

10. Pseudo functions

- 10.1 .ADDRSIZE
- 10.2 .BANK
- 10.3 .BANKBYTE
- 10.4 .BLANK
- 10.5 .CONCAT
- 10.6 .CONST
- 10.7 .HIBYTE
- 10.8 .HIWORD
- 10.9 .IDENT
- 10.10 .LEFT
- 10.11 .LOBYTE
- 10.12 .LOWORD
- 10.13 .MATCH
- 10.14 .MAX
- 10.15 .MID
- 10.16 .MIN
- 10.17 .REF, .REFERENCED
- 10.18 .RIGHT
- 10.19 .SIZEOF
- 10.20 .STRAT
- 10.21 .SPRINTF
- 10.22 .STRING
- 10.23 .STRLEN
- 10.24 .TCOUNT
- 10.25 .XMATCH

11. Control commands

- 11.1 .A16

- 11.2 .A8
- 11.3 .ADDR
- 11.4 .ALIGN
- 11.5 .ASCIIZ
- 11.6 .ASSERT
- 11.7 .AUTOIMPORT
- 11.8 .BANKBYTES
- 11.9 .BSS
- 11.10 .BYT, .BYTE
- 11.11 .CASE
- 11.12 .CHARMAP
- 11.13 .CODE
- 11.14 .CONDES
- 11.15 .CONSTRUCTOR
- 11.16 .DATA
- 11.17 .DBYT
- 11.18 .DEBUGINFO
- 11.19 .DEFINE
- 11.20 .DELMAC, .DELMACRO
- 11.21 .DEF, .DEFINED
- 11.22 .DEFINEDMACRO
- 11.23 .DESTRUCTOR
- 11.24 .DWORD
- 11.25 .ELSE
- 11.26 .ELSEIF
- 11.27 .END
- 11.28 .ENDENUM
- 11.29 .ENDIF
- 11.30 .ENDMAC, .ENDMACRO
- 11.31 .ENDPROC
- 11.32 .ENDREP, .ENDREPEAT
- 11.33 .ENDSCOPE
- 11.34 .ENDSTRUCT
- 11.35 .ENDUNION
- 11.36 .ENUM
- 11.37 .ERROR
- 11.38 .EXITMAC, .EXITMACRO
- 11.39 .EXPORT
- 11.40 .EXPORTZP
- 11.41 .FARADDR
- 11.42 .FATAL
- 11.43 .FEATURE
- 11.44 .FILEOPT, .FOPT
- 11.45 .FORCEIMPORT
- 11.46 .GLOBAL
- 11.47 .GLOBALZP
- 11.48 .HIBYTES
- 11.49 .I16
- 11.50 .I8
- 11.51 .IF
- 11.52 .IFBLANK
- 11.53 .IFCONST
- 11.54 .IFDEF
- 11.55 .IFNBLANK
- 11.56 .IFNDEF
- 11.57 .IFNREF
- 11.58 .IFP02
- 11.59 .IFP4510
- 11.60 .IFP816

- 11.61 .IFPC02
- 11.62 .IFPSC02
- 11.63 .IFREF
- 11.64 .IMPORT
- 11.65 .IMPORTZP
- 11.66 .INCBIN
- 11.67 .INCLUDE
- 11.68 .INTERRUPTOR
- 11.69 .ISMNEM, .ISMNEMONIC
- 11.70 .LINECONT
- 11.71 .LIST
- 11.72 .LISTBYTES
- 11.73 .LOBYTES
- 11.74 .LOCAL
- 11.75 .LOCALCHAR
- 11.76 .MACPACK
- 11.77 .MAC, .MACRO
- 11.78 .ORG
- 11.79 .OUT
- 11.80 .P02
- 11.81 .P4510
- 11.82 .P816
- 11.83 .PAGELEN, .PAGELENGTH
- 11.84 .PC02
- 11.85 .POPCPU
- 11.86 .POPSEG
- 11.87 .PROC
- 11.88 .PSC02
- 11.89 .PUSHCPU
- 11.90 .PUSHSEG
- 11.91 .RELOC
- 11.92 .REPEAT
- 11.93 .RES
- 11.94 .RODATA
- 11.95 .SCOPE
- 11.96 .SEGMENT
- 11.97 .SET
- 11.98 .SETCPU
- 11.99 .SMART
- 11.100 .STRUCT
- 11.101 .TAG
- 11.102 .UNDEF, .UNDEFINE
- 11.103 .UNION
- 11.104 .WARNING
- 11.105 .WORD
- 11.106 .ZEROPAGE

12. Macros

- 12.1 Introduction
- 12.2 Macros without parameters
- 12.3 Parametrized macros
- 12.4 Detecting parameter types
- 12.5 Recursive macros
- 12.6 Local symbols inside macros
- 12.7 C style macros
- 12.8 Characters in macros
- 12.9 Deleting macros

13. Macro packages

- 13.1 .MACPACK generic
- 13.2 .MACPACK longbranch
- 13.3 .MACPACK apple2
- 13.4 .MACPACK atari
- 13.5 .MACPACK cbm
- 13.6 .MACPACK cpu
- 13.7 .MACPACK module

14. Predefined constants

15. Structs and unions

- 15.1 Structs and unions Overview
- 15.2 Declaration
- 15.3 The .TAG keyword
- 15.4 Limitations

16. Module constructors/destructors

- 16.1 Module constructors/destructors Overview
- 16.2 Calling order
- 16.3 Pitfalls

17. Porting sources from other assemblers

- 17.1 TASS

18. Copyright

---

1. Overview

ca65 is a replacement for the ra65 assembler that was part of the cc65 C compiler, originally developed by John R. Dunning. I had some problems with ra65 and the copyright does not permit some things which I wanted to be possible, so I decided to write a completely new assembler/linker/archiver suite for the cc65 compiler. ca65 is part of this suite.

Some parts of the assembler (code generation and some routines for symbol table handling) are taken from an older crossassembler named a816 written by me a long time ago.

1.1 Design criteria

Here's a list of the design criteria, that I considered important for the development:

- The assembler must support macros. Macros are not essential, but they make some things easier, especially when you use the assembler in the backend of a compiler.
- The assembler must support the newer 65C02 and 65816 CPUs. I have been thinking about a 65816 backend for the C compiler, and even my old a816 assembler had support for these CPUs, so this wasn't really a problem.
- The assembler must produce relocatable code. This is necessary for the compiler support, and it is more convenient.

- Conditional assembly must be supported. This is a must for bigger projects written in assembler (like Elite128).
- The assembler must support segments, and it must support more than three segments (this is the count, most other assemblers support). Having more than one code segments helps developing code for systems with a divided ROM area (like the C64).
- The linker must be able to resolve arbitrary expressions. It should be able to get things like

```
.import S1, S2
.export Special
Special = 2*S1 + S2/7
```

  right.
- True lexical nesting for symbols. This is very convenient for larger assembly projects.
- "Cheap" local symbols without lexical nesting for those quick, late night hacks.
- I liked the idea of "options" as Anre Fachats .o65 format has it, so I introduced the concept into the object file format use by the new cc65 binutils.
- The assembler will be a one pass assembler. There was no real need for this decision, but I've written several multipass assemblers, and it started to get boring. A one pass assembler needs much more elaborated data structures, and because of that it's much more fun:-)
- Non-GPLed code that may be used in any project without restrictions or fear of "GPL infecting" other code.

2. Usage

2.1 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
  -d                            Debug mode
  -g                            Add debug info to object file
  -h                            Help (this text)
  -i                            Ignore case of symbols
  -l name                       Create a listing file if assembly was ok
  -mm model                     Set the memory model
  -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
  --bin-include-dir dir         Set a search path for binary includes
  --cpu type                    Set cpu type
  --create-dep name             Create a make dependency file
  --create-full-dep name        Create a full make dependency file
  --debug                       Debug mode
  --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
  --large-alignment             Don't warn about large alignments
  --listing name                Create a listing file if assembly was ok
  --list-bytes n                Maximum number of bytes per listing line
  --memory-model model          Set the memory model
  --pagelength n                Set the page length for the listing
  --relax-checks                Relax some checks (see docs)
```

```
  --smart                       Enable smart mode
  --target sys                  Set the target system
  --verbose                     Increase verbosity
  --version                     Print the assembler version
-------------------------------------------------------------------------------
```

2.2 Command line options in detail

Here is a description of all the command line options:

--bin-include-dir dir

Name a directory which is searched for binary 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 directories. See also the section about search paths.

--cpu type

Set the default for the CPU type. The option takes a parameter, which may be one of

6502, 6502X, 65SC02, 65C02, 65816, sweet16, HuC6280, 4510

--create-dep name

Tells the assembler to generate a file containing the dependency list for the assembled module in makefile syntax. The output is written to a file with the given name. The output does not include files passed via debug information to the assembler.

--create-full-dep name

Tells the assembler to generate a file containing the dependency list for the assembled module in makefile syntax. The output is written to a file with the given name. The output does include files passed via debug information to the assembler.

-d, --debug

Enables debug mode, something that should not be needed for mere mortals:-)

--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 overridden by the `.CASE` control command.

-l name, --listing name

Generate an assembler listing with the given name. A listing file will never be generated in case of assembly errors.

--large-alignment

Disable warnings about a large combined alignment. See the discussion of the .ALIGN directive for futher information.

--list-bytes n

Set the maximum number of bytes printed in the listing for one line of input. See the .LISTBYTES directive for more information. The value zero can be used to encode an unlimited number of printed bytes.

-mm model, --memory-model model

Define the default memory model. Possible model specifiers are near, far and huge.

-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.

--relax-checks

Relax some checks done by the assembler. This will allow code that is an error in most cases and flagged as such by the assembler, but can be valid in special situations.

Examples are:

- Short branches between two different segments.
- Byte sized address loads where the address is not a zeropage address.

-s, --smart-mode

In smart mode (enabled by -s or the .SMART pseudo instruction) the assembler will track usage of the REP and SEP instructions in 65816 mode and update the operand sizes accordingly. If the operand of such an instruction cannot be evaluated by the assembler (for example, because the operand is an imported symbol), a warning is issued.

Beware: Since the assembler cannot trace the execution flow this may lead to false results in some cases. If in doubt, use the .ixx and .axx instructions to tell the assembler about the current settings. Smart mode is off by default.

-t sys, --target sys

Set the target system. This will enable translation of character strings and character constants into the character set of the target platform. The default for the target system is "none", which means that no translation will take place. The assembler supports the same target systems as the compiler, see there for a list.

Depending on the target, the default CPU type is also set. This can be overriden by using the --cpu option.

-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 directories. See also the section about search paths.

-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.

3. Search paths

Normal include files are searched in the following places:

1. The current file's directory.
2. Any directory added with the `-I` option on the command line.
3. The value of the environment variable `CA65_INC` if it is defined.
4. A subdirectory named `asminc` of the directory defined in the environment variable `CC65_HOME`, if it is defined.
5. An optionally compiled-in directory.

Binary include files are searched in the following places:

1. The current file's directory.
2. Any directory added with the `--bin-include-dir` option on the command line.

4. Input format

4.1 Assembler syntax

The assembler accepts the standard 6502/65816 assembler syntax. One line may contain a label (which is identified by a colon), and, in addition to the label, an assembler mnemonic, a macro, or a control command (see section Control Commands for supported control commands). Alternatively, the line may contain a symbol definition using the '=' token. Everything after a semicolon is handled as a comment (that is, it is ignored).

Here are some examples for valid input lines:

```
Label:                          ; A label and a comment
        lda     #$20            ; A 6502 instruction plus comment
```

```
L1:     ldx     #$20            ; Same with label
L2:     .byte   "Hello world"   ; Label plus control command
        mymac   $20             ; Macro expansion
        MySym = 3*L1            ; Symbol definition
MaSym   = Label                 ; Another symbol
```

The assembler accepts

- all valid 6502 mnemonics when in 6502 mode (the default or after the `.P02` command was given).
- all valid 6502 mnemonics plus a set of illegal instructions when in 6502X mode.
- all valid 65SC02 mnemonics when in 65SC02 mode (after the `.PSC02` command was given).
- all valid 65C02 mnemonics when in 65C02 mode (after the `.PC02` command was given).
- all valid 65816 mnemonics when in 65816 mode (after the `.P816` command was given).
- all valid 4510 mnemonics when in 4510 mode (after the `.P4510` command was given).

4.2 65816 mode

In 65816 mode, several aliases are accepted, in addition to the official mnemonics:

- `CPA` is an alias for `CMP`
- `DEA` is an alias for `DEC A`
- `INA` is an alias for `INC A`
- `SWA` is an alias for `XBA`
- `TAD` is an alias for `TCD`
- `TAS` is an alias for `TCS`
- `TDA` is an alias for `TDC`
- `TSA` is an alias for `TSC`

4.3 6502X mode

6502X mode is an extension to the normal 6502 mode. In this mode, several mnemonics for illegal instructions of the NMOS 6502 CPUs are accepted. Since these instructions are illegal, there are no official mnemonics for them. The unofficial ones are taken from http://www.oxyron.de/html/opcodes02.html. Please note that only the ones marked as "stable" are supported. The following table uses information from the mentioned web page, for more information, see there.

- `ALR: A:=(A and #{imm})/2;`
- `ANC: A:=A and #{imm};` Generates opcode $0B.
- `ARR: A:=(A and #{imm})/2;`
- `AXS: X:=A and X-#{imm};`
- `DCP: {adr}:={adr}-1; A-{adr};`
- `ISC: {adr}:={adr}+1; A:=A-{adr};`
- `LAS: A,X,S:={adr} and S;`
- `LAX: A,X:={adr};`
- `RLA: {adr}:={adr}rol; A:=A and {adr};`
- `RRA: {adr}:={adr}ror; A:=A adc {adr};`
- `SAX: {adr}:=A and X;`
- `SLO: {adr}:={adr}*2; A:=A or {adr};`
- `SRE: {adr}:={adr}/2; A:=A xor {adr};`

4.4 4510 mode

The 4510 is a microcontroller that is the core of the Commodore C65 aka C64DX. It contains among other functions a slightly modified 65CE02/4502 CPU, to allow address mapping for 20 bits of address space (1 megabyte addressable area). As compared to the description of the CPU in the C65 System Specification (updated version) uses these changes:

- `LDA (d,SP),Y` may also be written as `LDA (d,S),Y` (matching the 65816 notataion).

- All branch instruction allow now 16 bit offsets. To use a 16 bit branch you have to prefix these with an "L" (e.g. "LBNE" instead of "BNE"). This might change at a later implementation of the assembler.

For more information about the Commodore C65/C64DX and the 4510 CPU, see http://www.zimmers.net/anonftp/pub/cbm/c65/ and Wikipedia.

4.5 sweet16 mode

SWEET 16 is an interpreter for a pseudo 16 bit CPU written by Steve Wozniak for the Apple][machines. It is available in the Apple][ROM. ca65 can generate code for this pseudo CPU when switched into sweet16 mode. The following is special in sweet16 mode:

- The '@' character denotes indirect addressing and is no longer available for cheap local labels. If you need cheap local labels, you will have to switch to another lead character using the .LOCALCHAR command.
- Registers are specified using R0 .. R15. In sweet16 mode, these identifiers are reserved words.

Please note that the assembler does neither supply the interpreter needed for SWEET 16 code, nor the zero page locations needed for the SWEET 16 registers, nor does it call the interpreter. All this must be done by your program. Apple][programmers do probably know how to use sweet16 mode.

For more information about SWEET 16, see http://www.6502.org/source/interpreters/sweet16.htm.

4.6 Number format

For literal values, the assembler accepts the widely used number formats: A preceding '$' or a trailing 'h' denotes a hex value, a preceding '%' denotes a binary value, and a bare number is interpreted as a decimal. There are currently no octal values and no floats.

4.7 Conditional assembly

Please note that when using the conditional directives (.IF and friends), the input must consist of valid assembler tokens, even in .IF branches that are not assembled. The reason for this behaviour is that the assembler must still be able to detect the ending tokens (like .ENDIF), so conversion of the input stream into tokens still takes place. As a consequence conditional assembly directives may **not** be used to prevent normal text (used as a comment or similar) from being assembled.

5. Expressions

5.1 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.

5.2 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, its 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.

5.3 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 extreme 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.

5.4 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.

5.5 Available operators

Operator	Description	Precedence
	Built-in string functions	0
	Built-in pseudo-variables	1
	Built-in pseudo-functions	1
+	Unary positive	1
-	Unary negative	1
~ .BITNOT	Unary bitwise not	1
< .LOBYTE	Unary low-byte operator	1
> .HIBYTE	Unary high-byte operator	1
^ .BANKBYTE	Unary bank-byte operator	1
*	Multiplication	2

/	Division	2
.MOD	Modulo operator	2
& .BITAND	Bitwise and	2
^ .BITXOR	Binary bitwise xor	2
<< .SHL	Shift-left operator	2
>> .SHR	Shift-right operator	2
+	Binary addition	3
-	Binary subtraction	3
| .BITOR	Bitwise or	3
=	Compare operator (equal)	4
<>	Compare operator (not equal)	4
<	Compare operator (less)	4
>	Compare operator (greater)	4
<=	Compare operator (less or equal)	4
>=	Compare operator (greater or equal)	4
&& .AND	Boolean and	5
.XOR	Boolean xor	5
|| .OR	Boolean or	6
! .NOT	Boolean not	7

Available operators, sorted by precedence

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

6. Symbols and labels

A symbol or label is an identifier that starts with a letter and is followed by letters and digits. Depending on some features enabled (see `at in identifiers`, `dollar in identifiers` and `leading dot in identifiers`) other characters may be present. Use of identifiers consisting of a single character will not work in all cases, because some of these identifiers are reserved keywords (for example "A" is not a valid identifier for a label, because it is the keyword for the accumulator).

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.

6.1 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 is almost identical, but causes the symbol to be marked as a label, so it may be handled differently in a debugger:

```
io := $d000
```

The right side can of course be an expression:

```
four = two * two
```

6.2 Numeric variables

Within macros and other control structures (.REPEAT, ...) it is sometimes useful to have some sort of variable. This can be achieved by the .SET operator. It creates a symbol that may get assigned a different value later:

```
four .set 4
lda     #four           ; Loads 4 into A
four .set 3
lda     #four           ; Loads 3 into A
```

Since the value of the symbol can change later, it must be possible to evaluate it when used (no delayed evaluation as with normal symbols). So the expression used as the value must be constant.

Following is an example for a macro that generates a different label each time it is used. It uses the .SPRINTF function and a numeric variable named lcount.

```
.lcount .set 0          ; Initialize the counter

.macro  genlab
        .ident (.sprintf ("L%04X", lcount)):
        lcount .set lcount + 1
.endmacro
```

6.3 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.

6.4 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.

6.5 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!
```

6.6 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.

Note: Scopes organize named symbols, not unnamed ones, so scopes don't have an effect on unnamed labels.

6.7 Using macros to define labels and constants

While there are drawbacks with this approach, it may be handy in a few rare situations. Using `.DEFINE`, it is possible to define symbols or constants that may be used elsewhere. One of the advantages is that you can use it to define string constants (this is not possible with the other symbol types).

Please note: `.DEFINE` style macros do token replacements on a low level, so the names do not adhere to scoping, diagnostics may be misleading, there are no symbols to look up in the map file, and there is no debug info. Especially the first problem in the list can lead to very nasty programming errors. Because of these problems, the general advice is, **NOT** do use `.DEFINE` if you don't have to.

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
```

6.8 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.

7. Scopes

ca65 implements several sorts of scopes for symbols.

7.1 Global scope

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

7.2 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.

7.3 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 precedes 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.

7.4 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
        ...
.endproc
```

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").

7.5 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.

7.6 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 preceded 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
```

7.7 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 preceding 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
```

8. Address sizes and memory models

8.1 Address sizes

ca65 assigns each segment and each symbol an address size. This is true, even if the symbol is not used as an address. You may also think of a value range of the symbol instead of an address size.

Possible address sizes are:

- Zeropage or direct (8 bits)
- Absolute (16 bits)
- Far (24 bits)
- Long (32 bits)

Since the assembler uses default address sizes for the segments and symbols, it is usually not necessary to override the default behaviour. In cases, where it is necessary, the following keywords may be used to specify address sizes:

- DIRECT, ZEROPAGE or ZP for zeropage addressing (8 bits).
- ABSOLUTE, ABS or NEAR for absolute addressing (16 bits).
- FAR for far addressing (24 bits).
- LONG or DWORD for long addressing (32 bits).

8.2 Address sizes of segments

The assembler assigns an address size to each segment. Since the representation of a label within this segment is "segment start + offset", labels will inherit the address size of the segment they are declared in.

The address size of a segment may be changed, by using an optional address size modifier. See the segment directive for an explanation on how this is done.

8.3 Address sizes of symbols

8.4 Memory models

The default address size of a segment depends on the memory model used. Since labels inherit the address size from the segment they are declared in, changing the memory model is an easy way to change the address size of many symbols at once.

9. Pseudo variables

Pseudo variables are readable in all cases, and in some special cases also writable.

9.1 *

Reading this pseudo variable will return the program counter at the start of the current input line.

Assignment to this variable is possible when .FEATURE pc_assignment is used. Note: You should not use assignments to *, use .ORG instead.

9.2 .ASIZE

Reading this pseudo variable will return the current size of the Accumulator in bits.

For the 65816 instruction set .ASIZE will return either 8 or 16, depending on the current size of the operand in immediate accu addressing mode.

For all other CPU instruction sets, .ASIZE will always return 8.

Example:

```
; Reverse Subtract with Accumulator
; A = memory - A
.macro rsb param
        .if .asize = 8
                eor     #$ff
        .else
                eor     #$ffff
        .endif
        sec
        adc     param
.endmacro
```

See also: .ISIZE

9.3 .CPU

Reading this pseudo variable will give a constant integer value that tells which CPU is currently enabled. It can also tell which instruction set the CPU is able to translate. The value read from the pseudo variable should be further examined by using one of the constants defined by the "cpu" macro package (see .MACPACK).

It may be used to replace the .IFPxx pseudo instructions or to construct even more complex expressions.

Example:

```
.macpack        cpu
.if     (.cpu .bitand CPU_ISET_65816)
        phx
        phy
.else
        txa
        pha
        tya
        pha
.endif
```

9.4 .ISIZE

Reading this pseudo variable will return the current size of the Index register in bits.

For the 65816 instruction set .ISIZE will return either 8 or 16, depending on the current size of the operand in immediate index addressing mode.

For all other CPU instruction sets, .ISIZE will always return 8.

See also: .ASIZE

9.5 .PARAMCOUNT

This builtin pseudo variable is only available in macros. It is replaced by the actual number of parameters that were given in the macro invocation.

Example:

```
.macro  foo     arg1, arg2, arg3
.if     .paramcount <> 3
.error  "Too few parameters for macro foo"
.endif
...
.endmacro
```

See section Macros.

9.6 .TIME

Reading this pseudo variable will give a constant integer value that represents the current time in POSIX standard (as seconds since the Epoch).

It may be used to encode the time of translation somewhere in the created code.

Example:

```
.dword  .time   ; Place time here
```

9.7 .VERSION

Reading this pseudo variable will give the assembler version according to the following formula:

VER_MAJOR*$100 + VER_MINOR*$10

It may be used to encode the assembler version or check the assembler for special features not available with older versions.

Example:

Version 2.14 of the assembler will return $2E0 as numerical constant when reading the pseudo variable .VERSION.

10. Pseudo functions

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

10.1 .ADDRSIZE

The .ADDRSIZE function is used to return the interal address size associated with a symbol. This can be helpful in macros when knowing the address size of symbol can help with custom instructions.

Example:

```
.macro myLDA foo
        .if .ADDRSIZE(foo) = 1
                ;do custom command based on zeropage addressing:
                .byte 0A5h, foo
        .elseif .ADDRSIZE(foo) = 2
                ;do custom command based on absolute addressing:
                .byte 0ADh
                .word foo
        .elseif .ADDRSIZE(foo) = 0
                ; no address size defined for this symbol:
                .out .sprintf("Error, address size unknown for symbol %s", .string(foo))
        .endif
.endmacro
```

This command is new and must be enabled with the .FEATURE addrsize command.

See: .FEATURE

10.2 .BANK

The .BANK function is used to support systems with banked memory. The argument is an expression with exactly one segment reference - usually a label. The function result is the value of the bank attribute assigned to the run memory area of the segment. Please see the linker documentation for more information about memory areas and their attributes.

The value of .BANK can be used to switch memory so that a memory bank containing specific data is available.

The bank attribute is a 32 bit integer and so is the result of the .BANK function. You will have to use .LOBYTE or similar functions to address just part of it.

Please note that .BANK will always get evaluated in the link stage, so an expression containing .BANK can never be used where a constant known result is expected (for example with .RES).

Example:

```
.segment "BANK1"
.proc   banked_func_1
        ...
.endproc

.segment "BANK2"
.proc   banked_func_2
        ...
.endproc
```

```
.proc   bank_table
        .addr   banked_func_1
        .byte   <.BANK (banked_func_1)

        .addr   banked_func_2
        .byte   <.BANK (banked_func_2)
.endproc
```

10.3 .BANKBYTE

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

See: .HIBYTE, .LOBYTE

10.4 .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. The token list that makes up the function argument may optionally be enclosed in curly braces. This allows the inclusion of tokens that would otherwise terminate the list (the closing right parenthesis). The curly braces are not considered part of the list, a list just consisting of curly braces is considered to be empty.

As an example, the .IFBLANK statement may be replaced by

```
.if     .blank({arg})
```

10.5 .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 concatenation 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"
```

10.6 .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)
```

10.7 .HIBYTE

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

See: .LOBYTE, .BANKBYTE

10.8 .HIWORD

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

See: .LOWORD

10.9 .IDENT

The function expects a string as its argument, and converts this argument into an identifier. If the string starts with the current .LOCALCHAR, it will be converted into a cheap local identifier, otherwise it will be converted into a normal identifier.

Example:

```
.macro  makelabel       arg1, arg2
        .ident (.concat (arg1, arg2)):
.endmacro

        makelabel       "foo", "bar"

        .word           foobar          ; Valid label
```

10.10 .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. The token list may optionally be enclosed into curly braces. This allows the inclusion of tokens that would otherwise terminate the list (the closing right paren in the given case).

Example:

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

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

        ; ldax called with immediate operand
        ...

        .endif
        ...
.endmacro
```

See also the .MID and .RIGHT builtin functions.

10.11 .LOBYTE

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

See: .HIBYTE, .BANKBYTE

10.12 .LOWORD

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

See: .HIWORD

10.13 .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

The token lists may optionally be enclosed into curly braces. This allows the inclusion of tokens that would otherwise terminate the list (the closing right paren in the given case). 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

10.14 .MAX

Builtin function. The result is the larger of two values.

The syntax is

```
.MAX (<value #1>, <value #2>)
```

Example:

```
; Reserve space for the larger of two data blocks
savearea:       .max (.sizeof (foo), .sizeof (bar))
```

See: .MIN

10.15 .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. The token list may optionally be enclosed into curly braces. This allows the inclusion of tokens that would otherwise terminate the list (the closing right paren in the given case).

Example:

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

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

        ; ldax called with immediate operand
        ...

        .endif
        ...
.endmacro
```

See also the .LEFT and .RIGHT builtin functions.

10.16 .MIN

Builtin function. The result is the smaller of two values.

The syntax is

```
.MIN (<value #1>, <value #2>)
```

Example:

```
; Reserve space for some data, but 256 bytes minimum
savearea:       .min (.sizeof (foo), 256)
```

See: .MAX

10.17 .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

10.18 .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. The token list may optionally be enclosed into curly braces. This allows the inclusion of tokens that would otherwise terminate the list (the closing right paren in the given case).

See also the .LEFT and .MID builtin functions.

10.19 .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 chosen over the symbol.

After the following code:

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

10.20 .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
```

10.21 .SPRINTF

Builtin function. It expects a format string as first argument. The number and type of the following arguments depend on the format string. The format string is similar to the one of the C `printf` function. Missing things are: Length modifiers, variable width.

The result of the function is a string.

Example:

```
num     = 3

; Generate an identifier:
.ident (.sprintf ("%s%03d", "label", num)):
```

10.22 .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
```

10.23 .STRLEN

Builtin function. The function accepts a string argument in braces and evaluates 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
```

10.24 .TCOUNT

Builtin function. The function accepts a token list in braces. The function result is the number of tokens given as argument. The token list may optionally be enclosed into curly braces which are not considered part of the list and not counted. Enclosement in curly braces allows the inclusion of tokens that would otherwise terminate the list (the closing right paren in the given case).

Example:

The `ldax` macro accepts the '#' token to denote immediate 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 immediate operand
        lda     #<(.right (.tcount ({arg})-1, {arg}))
        ldx     #>(.right (.tcount ({arg})-1, {arg}))
        .else
        ...
        .endif
.endmacro
```

10.25 .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

The token lists may optionally be enclosed into curly braces. This allows the inclusion of tokens that would otherwise terminate the list (the closing right paren in the given case). 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

11. Control commands

Here's a list of all control commands and a description, what they do:

11.1 .A16

Valid only in 65816 mode. Switch the accumulator to 16 bit.

Note: This command will not emit any code, it will tell the assembler to create 16 bit operands for immediate accumulator addressing mode.

See also: .SMART

11.2 .A8

Valid only in 65816 mode. Switch the accumulator to 8 bit.

Note: This command will not emit any code, it will tell the assembler to create 8 bit operands for immediate accu addressing mode.

See also: .SMART

11.3 .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

11.4 .ALIGN

Align data to a given boundary. The command expects a constant integer argument in the range 1 ... 65536, 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).

.ALIGN will insert fill bytes, and the number of fill bytes depend of the final address of the segment. .ALIGN cannot insert a variable number of bytes, since that would break address calculations within the module. So each .ALIGN expects the segment to be aligned to a multiple of the alignment, because that allows the number of

fill bytes to be calculated in advance by the assembler. You are therefore required to specify a matching alignment for the segment in the linker config. The linker will output a warning if the alignment of the segment is less than what is necessary to have a correct alignment in the object file.

Example:

```
.align  256
```

Some unexpected behaviour might occur if there are multiple `.ALIGN` commands with different arguments. To allow the assembler to calculate the number of fill bytes in advance, the alignment of the segment must be a multiple of each of the alignment factors. This may result in unexpectedly large alignments for the segment within the module.

Example:

```
.align  15
.byte   15
.align  18
.byte   18
```

For the assembler to be able to align correctly, the segment must be aligned to the least common multiple of 15 and 18 which is 90. The assembler will calculate this automatically and will mark the segment with this value.

Unfortunately, the combined alignment may get rather large without the user knowing about it, wasting space in the final executable. If we add another alignment to the example above

```
.align  15
.byte   15
.align  18
.byte   18
.align  251
.byte   0
```

the assembler will force a segment alignment to the least common multiple of 15, 18 and 251 - which is 22590. To protect the user against errors, the assembler will issue a warning when the combined alignment exceeds 256. The command line option `--large-alignment` will disable this warning.

Please note that with alignments that are a power of two (which were the only alignments possible in older versions of the assembler), the problem is less severe, because the least common multiple of powers to the same base is always the larger one.

11.5 .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).

11.6 .ASSERT

Add an assertion. The command is followed by an expression, an action specifier, and an optional message that is output in case the assertion fails. If no message was given, the string "Assertion failed" is used. The action specifier may be one of `warning`, `error`, `ldwarning` or `lderror`. In the former two cases, the assertion is

evaluated by the assembler if possible, and in any case, it's also passed to the linker in the object file (if one is generated). The linker will then evaluate the expression 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 of the output file is suppressed in both the assembler and linker.

11.7 .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 outputting 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
```

11.8 .BANKBYTES

Define byte sized data by extracting only the bank byte (that is, bits 16-23) from each expression. This is equivalent to .BYTE with the operator '^' prepended to each expression in its list.

Example:

```
.define MyTable TableItem0, TableItem1, TableItem2, TableItem3

TableLookupLo:   .lobytes   MyTable
TableLookupHi:   .hibytes   MyTable
TableLookupBank: .bankbytes MyTable
```

which is equivalent to

```
TableLookupLo:   .byte <TableItem0, <TableItem1, <TableItem2, <TableItem3
TableLookupHi:   .byte >TableItem0, >TableItem1, >TableItem2, >TableItem3
TableLookupBank: .byte ^TableItem0, ^TableItem1, ^TableItem2, ^TableItem3
```

See also: .BYTE, .HIBYTES, .LOBYTES

11.9 .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.

11.10 .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
```

11.11 .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
```

11.12 .CHARMAP

Apply a custom mapping for characters. The command is followed by two numbers. The first one is the index of the source character (range 0..255); the second one is the mapping (range 0..255). 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'
```

11.13 .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.

11.14 .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 specifying constructor and 1 is equal to specifying destructor). The .CONSTRUCTOR,

.DESTRUCTOR and .INTERRUPTOR commands are actually shortcuts for .CONDES with a type of constructor resp. destructor or interruptor.

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

11.15 .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.

11.16 .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.

11.17 .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.

11.18 .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
```

11.19 .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.

Please note that .DEFINE shares most disadvantages with its C counterpart, so the general advice is, **NOT** do use .DEFINE if you don't have to.

See also the .UNDEFINE command and section Macros.

11.20 .DELMAC, .DELMACRO

Delete a classic macro (defined with .MACRO) . The command is followed by the name of an existing macro. Its definition will be deleted together with the name. If necessary, another macro with this name may be defined later.

See: .ENDMACRO, .EXITMACRO, .MACRO

See also section Macros.

11.21 .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)
```

11.22 .DEFINEDMACRO

Builtin function. The function expects an identifier as argument in braces. The argument is evaluated, and the function yields "true" if the identifier has already been defined as the name of a macro. Otherwise the function yields false. Example:

```
.macro add foo
        clc
        adc foo
.endmacro

.if     .definedmacro(add)
        add #$01
.else
```

```
                clc
                adc #$01
        .endif
```

11.23 .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.

11.24 .DWORD

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

Example:

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

11.25 .ELSE

Conditional assembly: Reverse the current condition.

11.26 .ELSEIF

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

11.27 .END

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

11.28 .ENDENUM

End a .ENUM declaration.

11.29 .ENDIF

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

11.30 .ENDMAC, .ENDMACRO

Marks the end of a macro definition.

See: .DELMACRO, .EXITMACRO, .MACRO

See also section Macros.

11.31 .ENDPROC

End of local lexical level (see .PROC).

11.32 .ENDREP, .ENDREPEAT

End a .REPEAT block.

11.33 .ENDSCOPE

End of local lexical level (see .SCOPE).

11.34 .ENDSTRUCT

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

11.35 .ENDUNION

Ends a union definition. See the .UNION command and the separate section named "Structs and unions".

11.36 .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 preceding plus one. This behaviour may be overridden 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.

11.37 .ERROR

Force an assembly error. The assembler will output an error message preceded by "User error". Assembly is continued but no object file will generated.

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: .FATAL, .OUT, .WARNING

11.38 .EXITMAC, .EXITMACRO

Abort a macro expansion immediately. This command is often useful in recursive macros.

See: .DELMACRO, .ENDMACRO, .MACRO

See also section Macros.

11.39 .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 and (also optional) an assignment. Using an additional assignment in the export statement allows to define and export a symbol in one statement. 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.

Examples:

```
.export foo
.export bar: far
.export foobar: far = foo * bar
```

```
.export baz := foobar, zap: far = baz - bar
```

As with constant definitions, using := instead of = marks the symbols as a label.

See: .EXPORTZP

11.40 .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 symbols. An assignment may be included in the .EXPORTZP statement. This allows to define and export a symbol in one statement.

Examples:

```
.exportzp  foo, bar
.exportzp  baz := $02
```

See: .EXPORT

11.41 .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

11.42 .FATAL

Force an assembly error and terminate assembly. The assembler will output an error message preceded by "User error" and will terminate assembly immediately.

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
.fatal  "Must define foo or bar!"
.endif
```

See also: .ERROR, .OUT, .WARNING

11.43 .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

```
.FEATURE        xxx
```

will enable the feature until end of assembly is reached.

The following features are available:

addrsize

Enables the .ADDRSIZE pseudo function. This function is experimental and not enabled by default.

See also: .ADDRSIZE

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.

bracket_as_indirect

Use `[]` instead of `()` for the indirect addressing modes. Example:

```
        lda     [$82]
        lda     [$82,x]
        lda     [$82],y
        jmp     [$fffe]
        jmp     [table,x]
```

Note: This should not be used in 65186 mode because it conflicts with the 65816 instruction syntax for far addressing. See the section covering address sizes for more information.

c_comments

Allow C like comments using `/*` and `*/` as left and right comment terminators. Note that C comments may not be nested. There's also a pitfall when using C like comments: All statements must be terminated by "end-of-line". Using C like comments, it is possible to hide the newline, which results in error messages. See the following non working example:

```
        lda     #$00  /* This comment hides the newline
*/      sta     $82
```

dollar_in_identifiers

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

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.

force_range

Force expressions into their valid range for immediate addressing and storage operators like .BYTE and .WORD. Be very careful with this one, since it will completely disable error checks.

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).

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.

loose_char_term

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

loose_string_term

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

missing_char_term

Accept single quoted character constants where the terminating quote is missing.

```
lda     #'a
```

Note: This does not work in conjunction with `.FEATURE loose_string_term`, since in this case the input would be ambiguous.

org_per_seg

This feature makes relocatable/absolute mode local to the current segment. Using `.ORG` when `org_per_seg` is in effect will only enable absolute mode for the current segment. Dito for `.RELOC`.

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).

ubiquitous_idents

Allow the use of instructions names as names for macros and symbols. This makes it possible to "overload" instructions by defining a macro with the same name. This does also make it possible to introduce hard to find errors in your code, so be careful!

underline_in_numbers

Allow underlines within numeric constants. These may be used for grouping the digits of numbers for easier reading. Example:

```
.feature        underline_in_numbers
.word           %1100001110100101
.word           %1100_0011_1010_0101    ; Identical but easier to read
```

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.

11.44 .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"
```

11.45 .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

11.46 .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
```

11.47 .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 symbols.

Example:

```
.globalzp foo, bar
```

11.48 .HIBYTES

Define byte sized data by extracting only the high byte (that is, bits 8-15) from each expression. This is equivalent to .BYTE with the operator '>' prepended to each expression in its list.

Example:

```
.lobytes        $1234, $2345, $3456, $4567
.hibytes        $fedc, $edcb, $dcba, $cba9
```

which is equivalent to

```
.byte           $34, $45, $56, $67
.byte           $fe, $ed, $dc, $cb
```

Example:

```
.define MyTable TableItem0, TableItem1, TableItem2, TableItem3

TableLookupLo:   .lobytes MyTable
TableLookupHi:   .hibytes MyTable
```

which is equivalent to

```
TableLookupLo:   .byte <TableItem0, <TableItem1, <TableItem2, <TableItem3
TableLookupHi:   .byte >TableItem0, >TableItem1, >TableItem2, >TableItem3
```

See also: .BYTE, .LOBYTES, .BANKBYTES

11.49 .I16

Valid only in 65816 mode. Switch the index registers to 16 bit.

Note: This command will not emit any code, it will tell the assembler to create 16 bit operands for immediate operands.

See also the .I8 and .SMART commands.

11.50 .I8

Valid only in 65816 mode. Switch the index registers to 8 bit.

Note: This command will not emit any code, it will tell the assembler to create 8 bit operands for immediate operands.

See also the .I16 and .SMART commands.

11.51 .IF

Conditional assembly: Evaluate 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.

11.52 .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 .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 TRUE if an empty parameter was given.

Example:

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

See also: .BLANK

11.53 .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

11.54 .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

11.55 .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

11.56 .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

11.57 .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

11.58 .IFP02

Conditional assembly: Check if the assembler is currently in 6502 mode (see .P02 command).

11.59 .IFP4510

Conditional assembly: Check if the assembler is currently in 4510 mode (see .P4510 command).

11.60 .IFP816

Conditional assembly: Check if the assembler is currently in 65816 mode (see .P816 command).

11.61 .IFPC02

Conditional assembly: Check if the assembler is currently in 65C02 mode (see .PC02 command).

11.62 .IFPSC02

Conditional assembly: Check if the assembler is currently in 65SC02 mode (see .PSC02 command).

11.63 .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

11.64 .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

11.65 .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

11.66 .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 assumed (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
```

11.67 .INCLUDE

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

Example:

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

11.68 .INTERRUPTOR

Export a symbol and mark it as an interruptor. This may be used together with the linker to build a table of interruptor subroutines that are called in an interrupt.

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 interruptor does nothing by itself.

An interruptor is always exported as an absolute (16 bit) symbol. You don't need to use an additional `.export` statement, this is implied by `.interruptor`. 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:

```
.interruptor    IrqHandler
.interruptor    Handler, 16
```

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

11.69 .ISMNEM, .ISMNEMONIC

Builtin function. The function expects an identifier as argument in braces. The argument is evaluated, and the function yields "true" if the identifier is defined as an instruction mnemonic that is recognized by the assembler. Example:

```
.if     .not .ismnemonic(ina)
        .macro ina
                clc
                adc #$01
        .endmacro
.endif
```

11.70 .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
```

11.71 .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
```

11.72 .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
```

11.73 .LOBYTES

Define byte sized data by extracting only the low byte (that is, bits 0-7) from each expression. This is equivalent to .BYTE with the operator '<' prepended to each expression in its list.

Example:

```
.lobytes        $1234, $2345, $3456, $4567
.hibytes        $fedc, $edcb, $dcba, $cba9
```

which is equivalent to

```
.byte           $34, $45, $56, $67
.byte           $fe, $ed, $dc, $cb
```

Example:

```
.define MyTable TableItem0, TableItem1, TableItem2, TableItem3

TableLookupLo:   .lobytes MyTable
TableLookupHi:   .hibytes MyTable
```

which is equivalent to

```
TableLookupLo:   .byte <TableItem0, <TableItem1, <TableItem2, <TableItem3
TableLookupHi:   .byte >TableItem0, >TableItem1, >TableItem2, >TableItem3
```

See also: .BYTE, .HIBYTES, .BANKBYTES

11.74 .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.

11.75 .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!
```

11.76 .MACPACK

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

```
atari           Defines the scrcode macro.
cbm             Defines the scrcode macro.
cpu             Defines constants for the .CPU variable.
generic         Defines generic macroes like add, sub, and blt.
longbranch      Defines conditional long-jump macroes.
```

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.

11.77 .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. A macro definition is terminated by .ENDMACRO.

Example:

```
.macro  ldax    arg             ; Define macro ldax
        lda     arg
        ldx     arg+1
```

See: .DELMACRO, .ENDMACRO, .EXITMACRO

See also section Macros.

11.78 .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.

By default, absolute/relocatable mode is global (valid even when switching segments). Using .FEATURE org_per_seg it can be made segment local.

Please note that you *do not need* .ORG 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.

Example:

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

11.79 .OUT

Output a string to the console without producing an error. This command is similar 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: .ERROR, .FATAL, .WARNING

11.80 .P02

Enable the 6502 instruction set, disable 65SC02, 65C02 and 65816 instructions. This is the default if not overridden by the --cpu command line option.

See: .PC02, .PSC02, .P816 and .P4510

11.81 .P4510

Enable the 4510 instruction set. This is a superset of the 65C02 and 6502 instruction sets.

See: .P02, .PSC02, .PC02 and .P816

11.82 .P816

Enable the 65816 instruction set. This is a superset of the 65SC02 and 6502 instruction sets.

See: .P02, .PSC02, .PC02 and .P4510

11.83 .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
```

11.84 .PC02

Enable the 65C02 instructions set. This instruction set includes all 6502 and 65SC02 instructions.

See: .P02, .PSC02, .P816 and .P4510

11.85 .POPCPU

Pop the last CPU setting from the stack, and activate it.

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

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

See: .CPU, .PUSHCPU, .SETCPU

11.86 .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

11.87 .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

11.88 .PSC02

Enable the 65SC02 instructions set. This instruction set includes all 6502 instructions.

See: .P02, .PC02, .P816 and .P4510

11.89 .PUSHCPU

Push the currently active CPU onto a stack. The stack has a size of 8 entries.

.PUSHCPU allows together with .POPCPU to switch to another CPU and to restore the old CPU later, without knowledge of the current CPU setting.

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

See: .CPU, .POPCPU, .SETCPU

11.90 .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

11.91 .RELOC

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

11.92 .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) ^ $55
        .endrep
.endmacro
```

See: .ENDREPEAT

11.93 .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
```

11.94 .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.

11.95 .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
.endscope                       ; Close lexical level

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

See: .ENDSCOPE and .PROC

11.96 .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 ("ZEROPAGE", "CODE", "RODATA", "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 address size separated by a colon. See the section covering address sizes for more information.

The default address size for a segment depends on the memory model specified on the command line. The default is "absolute", which means that you don't have to use an address size modifier in most cases.

"absolute" means that the is a segment with 16 bit (absolute) addressing. That is, the segment will reside somewhere in core memory outside the zero page. "zeropage" (8 bit) means that 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.

Please note that a segment cannot have two different address sizes. A segment specified as zeropage cannot be declared as being absolute later.

Examples:

```
.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, .RODATA, and .ZEROPAGE

11.97 .SET

.SET is used to assign a value to a variable. See Numeric variables for a full description.

11.98 .SETCPU

Switch the CPU instruction set. The command is followed by a string that specifies the CPU. Possible values are those that can also be supplied to the --cpu command line option, namely: 6502, 6502X, 65SC02, 65C02, 65816, 4510 and HuC6280.

See: .CPU, .IFP02, .IFP816, .IFPC02, .IFPSC02, .P02, .P816, .P4510, .PC02, .PSC02

11.99 .SMART

Switch on or off smart mode. The command must be followed by a '+' or '-' character to switch the option on or off respectively. The default is off (that is, the assembler doesn't try to be smart), but this default may be changed by the -s switch on the command line.

In smart mode the assembler will do the following:

- Track usage of the REP and SEP instructions in 65816 mode and update the operand sizes accordingly. If the operand of such an instruction cannot be evaluated by the assembler (for example, because the operand is an imported symbol), a warning is issued. Beware: Since the assembler cannot trace the execution flow this may lead to false results in some cases. If in doubt, use the .Inn and .Ann instructions to tell the assembler about the current settings.
- In 65816 mode, replace a RTS instruction by RTL if it is used within a procedure declared as far, or if the procedure has no explicit address specification, but it is far because of the memory model used.

Example:

```
.smart                          ; Be smart
.smart  -                       ; Stop being smart
```

See: .A16, .A8, .I16, .I8

11.100 .STRUCT

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

See also: .ENDSTRUCT, .ENDUNION, .UNION

11.101 .TAG

Allocate space for a struct or union.

Example:

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

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

11.102 .UNDEF, .UNDEFINE

Delete a define style macro definition. The command is followed by an identifier which specifies the name of the macro to delete. Macro replacement is switched of when reading the token following the command (otherwise the macro name would be replaced by its replacement list).

See also the .DEFINE command and section Macros.

11.103 .UNION

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

See also: .ENDSTRUCT, .ENDUNION, .STRUCT

11.104 .WARNING

Force an assembly warning. The assembler will output a warning message preceded 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: .ERROR, .FATAL, .OUT

11.105 .WORD

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

Example:

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

11.106 .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

12. Macros

12.1 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.

12.2 Macros without parameters

In its 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
```

12.3 Parametrized macros

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

```
.macro  inc16   addr
        clc
        lda     addr
        adc     #<$0001
        sta     addr
        lda     addr+1
        adc     #>$0001
        sta     addr+1
.endmacro
```

When calling the macro, you may give a parameter, and each occurrence 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     #<$0001
        sta     $1000
        lda     $1000+1
        adc     #>$0001
        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 expanded). 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
```

That 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. That command is replaced by the parameter count given, *including* explicitly empty 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
```

Macro parameters may optionally be enclosed into curly braces. This allows the inclusion of tokens that would otherwise terminate the parameter (the comma in case of a macro parameter).

```
.macro  foo     arg1, arg2
        ...
.endmacro

        foo     ($00,x)         ; Two parameters passed
        foo     {($00,x)}       ; One parameter passed
```

In the first case, the macro is called with two parameters: '`($00`' and '`x)`'. The comma is not passed to the macro, because it is part of the calling sequence, not the parameters.

In the second case, '`($00,x)`' is passed to the macro; this time, including the comma.

12.4 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. Please note how the curly braces are used to enclose parameters to pseudo functions handling token lists. This is necessary, because the token lists may include commas or parens, which would be treated by the assembler as end-of-list.

The macro can be used as

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

12.5 Recursive macros

Macros may be used recursively:

```
.macro  push    r1, r2, r3
        lda     r1
        pha
.ifnblank       r2
        push    r2, r3
.endif
.endmacro
```

There's also a special macro command to help with writing recursive macros: .EXITMACRO. That command will stop macro expansion immediately:

```
.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 that 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
```

12.6 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     #<$0001
        sta     addr
        lda     addr+1
        adc     #>$0001
        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 possible solution is the command .LOCAL. It 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 can solve the problem above by using .LOCAL:

```
.macro  inc16   addr
        .local  Skip            ; Make Skip a local symbol
        inc     addr
        bne     Skip
        inc     addr+1
Skip:                           ; Not visible outside
.endmacro
```

Another solution is of course to start a new lexical block inside the macro that hides any labels:

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

12.7 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.
- Parentheses work differently from C macros. The common practice of wrapping C macros in parentheses may cause unintended problems here, such as accidentally implying an indirect addressing mode. While the definition of a macro requires parentheses around its argument list, when invoked they should not be included.

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 parentheses, the actual argument used when invoking the macro should not be. The invoked arguments are separated by commas only, if parentheses are used by accident they will become part of the replaced token.

If you wish to have an expression follow the macro invocation, the last parameter can be enclosed in curly braces {} to indicate the end of that argument.

Examples:

```
.define COMBINE(ta,tb,tc) ta+tb*10+tc*100

.word COMBINE 5,6,7       ; 5+6*10+7*100 = 765
.word COMBINE(5,6,7)      ; (5+6*10+7)*100 = 7200 ; incorrect use of parentheses
.word COMBINE 5,6,7+1     ; 5+6*10+7+1*100 = 172
.word COMBINE 5,6,{7}+1   ; 5+6*10+7*100+1 = 766 ; {} encloses the argument
.word COMBINE 5,6-2,7     ; 5+6-2*10+7*100 = 691
.word COMBINE 5,(6-2),7   ; 5+(6-2)*10+7*100 = 745
.word COMBINE 5,6,7+COMBINE 0,1,2     ; 5+6*10+7+0+1*10+2*100*100 = 20082
.word COMBINE 5,6,{7}+COMBINE 0,1,2   ; 5+6*10+7*100+0+1*10+2*100 = 975
```

With C macros it is common to enclose the results in parentheses to prevent unintended interactions with the text of the arguments, but additional care must be taken in this assembly context where parentheses may alter the meaning of a statement. In particular, indirect addressing modes may be accidentally implied:

```
.define DUO(ta,tb) (ta+(tb*10))

        lda DUO(5,4), Y         ; LDA (indirect), Y
        lda 0+DUO(5,4), Y       ; LDA absolute indexed, Y
```

12.8 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.

12.9 Deleting macros

Macros can be deleted. This will not work if the macro that should be deleted is currently expanded as in the following non-working example:

```
.macro  notworking
        .delmacro       notworking
.endmacro

        notworking              ; Will not work
```

The commands to delete classic and define style macros differ. Classic macros can be deleted by use of .DELMACRO, while for .DEFINE style macros, .UNDEFINE must be used. Example:

```
.define value   1
.macro  mac
        .byte   2
.endmacro

        .byte   value           ; Emit one byte with value 1
        mac                     ; Emit another byte with value 2

.undefine value
.delmacro mac

        .byte   value           ; Error: Unknown identifier
        mac                     ; Error: Missing ":"
```

A separate command for .DEFINE style macros was necessary, because the name of such a macro is replaced by its replacement list on a very low level. To get the actual name, macro replacement has to be switched off when reading the argument to .UNDEFINE. This does also mean that the argument to .UNDEFINE is not allowed to come from another .DEFINE. All this is not necessary for classic macros, so having two different commands increases flexibility.

13. Macro packages

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

13.1 .MACPACK generic

This macro package defines macroes that are useful in almost any program. Currently defined macroes are:

```
        .macro  add     Arg     ; add without carry
                clc
                adc     Arg
        .endmacro

        .macro  sub     Arg     ; subtract without borrow
                sec
                sbc     Arg
        .endmacro

        .macro  bge     Arg     ; branch on greater-than or equal
                bcs     Arg
        .endmacro
```

```
.macro  blt     Arg     ; branch on less-than
        bcc     Arg
.endmacro

.macro  bgt     Arg     ; branch on greater-than
        .local  L
        beq     L
        bcs     Arg
L:
.endmacro

.macro  ble     Arg     ; branch on less-than or equal
        beq     Arg
        bcc     Arg
.endmacro

.macro  bnz     Arg     ; branch on not zero
        bne     Arg
.endmacro

.macro  bze     Arg     ; branch on zero
        beq     Arg
.endmacro
```

13.2 .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
```

13.3 .MACPACK apple2

This macro package defines a macro named scrcode. It takes a string as argument and places this string into memory translated into screen codes.

13.4 .MACPACK atari

This macro package defines a macro named scrcode. It takes a string as argument and places this string into memory translated into screen codes.

13.5 .MACPACK cbm

This macro package defines a macro named scrcode. It takes a string as argument and places this string into memory translated into screen codes.

13.6 .MACPACK cpu

This macro package does not define any macros but constants used to examine the value read from the .CPU pseudo variable. For each supported CPU a constant similar to

```
CPU_6502
CPU_65SC02
CPU_65C02
CPU_65816
CPU_SWEET16
CPU_HUC6280
CPU_4510
```

is defined. These constants may be used to determine the exact type of the currently enabled CPU. In addition to that, for each CPU instruction set, another constant is defined:

```
CPU_ISET_6502
CPU_ISET_65SC02
CPU_ISET_65C02
CPU_ISET_65816
CPU_ISET_SWEET16
CPU_ISET_HUC6280
CPU_ISET_4510
```

The value read from the .CPU pseudo variable may be checked with .BITAND to determine if the currently enabled CPU supports a specific instruction set. For example the 65C02 supports all instructions of the 65SC02 CPU, so it has the CPU_ISET_65SC02 bit set in addition to its native CPU_ISET_65C02 bit. Using

```
.if (.cpu .bitand CPU_ISET_65SC02)
        lda     (sp)
.else
        ldy     #$00
        lda     (sp),y
.endif
```

it is possible to determine if the

```
        lda     (sp)
```

instruction is supported, which is the case for the 65SC02, 65C02 and 65816 CPUs (the latter two are upwards compatible to the 65SC02).

13.7 .MACPACK module

This macro package defines a macro named module_header. It takes an identifier as argument and is used to define the header of a module both in the dynamic and static variant.

14. Predefined constants

For better orthogonality, the assembler defines similar symbols as the compiler, depending on the target system selected:

- __APPLE2__ - Target system is apple2 or apple2enh
- __APPLE2ENH__ - Target system is apple2enh
- __ATARI2600__ - Target system is atari2600
- __ATARI5200__ - Target system is atari5200
- __ATARI__ - Target system is atari or atarixl
- __ATARIXL__ - Target system is atarixl
- __ATMOS__ - Target system is atmos
- __BBC__ - Target system is bbc
- __C128__ - Target system is c128

- __C16__ - Target system is c16 or plus4
- __C64__ - Target system is c64
- __CBM__ - Target is a Commodore system
- __CBM510__ - Target system is cbm510
- __CBM610__ - Target system is cbm610
- __GEOS__ - Target is a GEOS system
- __GEOS_APPLE__ - Target system is geos-apple
- __GEOS_CBM__ - Target system is geos-cbm
- __LUNIX__ - Target system is lunix
- __LYNX__ - Target system is lynx
- __NES__ - Target system is nes
- __OSIC1P__ - Target system is osic1p
- __PET__ - Target system is pet
- __PLUS4__ - Target system is plus4
- __SIM6502__ - Target system is sim6502
- __SIM65C02__ - Target system is sim65c02
- __SUPERVISION__ - Target system is supervision
- __VIC20__ - Target system is vic20

15. Structs and unions

15.1 Structs and unions Overview

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.

15.2 Declaration

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. The offset of all members relative to the union is zero.

```
.union  Entry
        index   .word
        ptr     .addr
.endunion
```

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
```

15.3 The .TAG keyword

Using the .TAG keyword, it is possible to reserve space for an already defined struct or unions within another struct:

```
.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.

15.4 Limitations

Structs and unions are currently implemented as nested symbol tables (in fact, they were a by-product of the improved scoping rules). Currently, the assembler has no idea of types. This means that the .TAG keyword will only allocate space. You won't be able to initialize variables declared with .TAG, and adding an embedded structure to another structure with .TAG will not make this structure accessible by using the '::' operator.

16. 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.

16.1 Module constructors/destructors Overview

Using the `.CONSTRUCTOR`, `.DESTRUCTOR` and `.INTERRUPTOR` keywords it is 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, or a table of interrupt handler functions.

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).

16.2 Calling order

The symbols 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 function 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, functions are called with high priority functions first, followed by low priority functions.

16.3 Pitfalls

When using these special symbols, 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` and `callirq` modules 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 or an irq handler is needed for a group of functions, be sure to place the 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 initialization or cleanup code does depend on other initialization or cleanup code, you have to choose the priority for the functions accordingly.
- Besides the `.CONSTRUCTOR`, `.DESTRUCTOR` and `.INTERRUPTOR` statements, there is also a more generic command: `.CONDES`. This allows to specify an additional type. Predefined types are 0 (constructor), 1 (destructor) and 2 (interruptor). The linker generates a separate table for each type on request.

17. Porting sources from other assemblers

Sometimes it is necessary to port code written for older assemblers to ca65. In some cases, this can be done without any changes to the source code by using the emulation features of ca65 (see `.FEATURE`). In other cases, it is necessary to make changes to the source code.

Probably the biggest difference is the handling of the `.ORG` directive. ca65 generates relocatable code, and placement is done by the linker. Most other assemblers generate absolute code, placement is done within the assembler and there is no external linker.

In general it is not a good idea to write new code using the emulation features of the assembler, but there may be situations where even this rule is not valid.

17.1 TASS

You need to use some of the ca65 emulation features to simulate the behaviour of such simple assemblers.

1. Prepare your sourcecode like this:

```
; if you want TASS style labels without colons
.feature labels_without_colons

; if you want TASS style character constants
; ("a" instead of the default 'a')
.feature loose_char_term

        .word *+2       ; the cbm load address

        [yourcode here]
```

notice that the two emulation features are mostly useful for porting sources originally written in/for TASS, they are not needed for the actual "simple assembler operation" and are not recommended if you are writing new code from scratch.

2. Replace all program counter assignments (which are not possible in ca65 by default, and the respective emulation feature works different from what you'd expect) by another way to skip to memory locations, for example the .RES directive.

```
; *=$2000
.res $2000-*    ; reserve memory up to $2000
```

Please note that other than the original TASS, ca65 can never move the program counter backwards - think of it as if you are assembling to disk with TASS.

3. Conditional assembly (.ifeq/.endif/.goto etc.) must be rewritten to match ca65 syntax. Most importantly notice that due to the lack of .goto, everything involving loops must be replaced by .REPEAT.
4. To assemble code to a different address than it is executed at, use the .ORG directive instead of .offs-constructs.

```
.org $1800

[floppy code here]

.reloc  ; back to normal
```

5. Then assemble like this:

```
cl65 --start-addr 0x0ffe -t none myprog.s -o myprog.prg
```

Note that you need to use the actual start address minus two, since two bytes are used for the cbm load address.

18. Copyright