ACME 0.97 – User and Language Reference

ACME Crossassembler for Multiple Environments

This is the combined Markdown edition of the `docs/` directory from the local Windows package.
The editor's quick-start section and the original English documentation are presented together. This reference applies to the bundled ACME version and may not describe the latest upstream release.

C64 Quick Start

Run the following commands from the `acme.exe` directory in PowerShell. Replace `main.a` with the name of your source file.

```
.\acme.exe -f cbm -o main.prg main.a
```

The `cbm` format creates a PRG with a two-byte load address. The assembler does not start the program automatically.

Minimal C64 example with a `$C000` start address:

```
!cpu 6502
* = $c000

    lda #$06
    sta $d020
    rts
```

After loading the PRG, `SYS 49152` starts the code. Setting the `$0801` start address alone does not create a BASIC SYS stub.

Common Operations

Feladat	ACME-szintaxis
Start address	`* = $c000`
Bytes	`!byte $01, $02, $ff`
16-bit values	`!word $0400, $2000`
Raw text data	`!text "HELLO", 0`
PETSCII-adatok	`!pet "HELLO", 13, 0`

Feladat	ACME-szintaxis
Screen-code data	`!scr "HELLO"`
Fill	`!fill 64, 0`
Align to a page boundary	`!align 255, 0`
Include source	`!source "routines.a"`
Include binary data	`!binary "sprite.bin"`
Constant	`SCREEN = $0400`
Local namespace	`!zone name { ... }`
Define / call a macro	`!macro name .arg { ... }` / `+name $06`
6510 undocumented instructions	`!cpu 6510` – see the undocumented opcodes section

The first argument of `!align` is a bit mask and the second is the desired masked value; it is not a simple byte-boundary parameter. See the directive reference for details and aliases.

ACME and Visual Assembler

ACME's `!text` directive stores data; it does not print text at screen coordinates.
The VA forms `!text x, y, "text", lower`, `.clear_screen`, `!charset lower`, and `!region` / `!endregion` are VA extensions and must not be treated as standalone ACME source automatically. VA must expand them before an external ACME build.

Text after `;` is a comment. A symbol beginning with `.` can be a local label in ACME; not every dot-prefixed name is a directive.

Table of Contents

- Quick Reference — `QuickRef.txt`
- Complete Directive Reference — `AllPOs.txt`
- Addressing Modes and Size Forcing — `AddrModes.txt`
- Errors and Solutions — `Errors.txt`
- Undocumented Opcodes — `Illegals.txt`
- Floating-Point Expressions — `Floats.txt`
- 65816-Specific Behavior — `65816.txt`
- General Description and Platforms — `Help.txt`
- Macro Library — `Lib.txt`
- Building the Package Examples — `Example.txt`
- Building ACME from Source — `Source.txt`
- Compatibility and Version Changes — `Upgrade.txt`
- Change Log — `Changes.txt`
- CPU: all — `cputypes/all.txt`
- CPU: 4502 — `cputypes/cpu 4502.txt`
- CPU: 6502 — `cputypes/cpu 6502.txt`
- CPU: 65816 — `cputypes/cpu 65816.txt`
- CPU: 65c02 — `cputypes/cpu 65c02.txt`

- CPU: 65ce02 — `cputypes/cpu 65ce02.txt`
- CPU: m65 — `cputypes/cpu m65.txt`
- CPU: nmos6502 — `cputypes/cpu nmos6502.txt`
- CPU: r65c02 — `cputypes/cpu r65c02.txt`
- CPU: w65c02 — `cputypes/cpu w65c02.txt`
- License – GNU GPL v2 — `COPYING`

Directive Index

The call forms are listed in the original order used by the detailed reference; aliases are shown in the corresponding entry.

- `!8 EXPRESSION [, EXPRESSION]*`
- `!16 EXPRESSION [, EXPRESSION]*`
- `!le16 EXPRESSION [, EXPRESSION]*`
- `!be16 EXPRESSION [, EXPRESSION]*`
- `!24 EXPRESSION [, EXPRESSION]*`
- `!le24 EXPRESSION [, EXPRESSION]*`
- `!be24 EXPRESSION [, EXPRESSION]*`
- `!32 EXPRESSION [, EXPRESSION]*`
- `!le32 EXPRESSION [, EXPRESSION]*`
- `!be32 EXPRESSION [, EXPRESSION]*`
- `!hex PAIRS_OF_HEX_DIGITS`
- `!fill AMOUNT [, VALUE]`
- `!skip AMOUNT`
- `!align ANDVALUE, EQUALVALUE [, FILLVALUE]`
- `!convtab KEYWORD [ { BLOCK } ]`
- `!text STRING_VALUE [, STRING_VALUE]*`
- `!pet STRING_VALUE [, STRING_VALUE]*`
- `!raw STRING_VALUE [, STRING_VALUE]*`
- `!scr STRING_VALUE [, STRING_VALUE]*`
- `!scrxor XOR_VALUE, STRING_VALUE [, STRING_VALUE]*`
- `!to FILENAME, FILEFORMAT`
- `!source FILENAME`
- `!binary FILENAME [, [SIZE] [, [SKIP]]]`
- `!zone [TITLE] [ { BLOCK } ]`
- `!symbollist FILENAME`
- `!if CONDITION { BLOCK } [ else { BLOCK } ]`
- `!ifdef SYMBOL { BLOCK } [ else { BLOCK } ]`
- `!ifndef SYMBOL { BLOCK } [ else { BLOCK } ]`
- `!for SYMBOL, START, END { BLOCK }`
- `!set SYMBOL = VALUE`
- `!do [KEYWORD CONDITION] { BLOCK } [KEYWORD CONDITION]`
- `!while [CONDITION] { BLOCK }`
- `!endoffile`
- `!warn VALUE [, VALUE]*`
- `!error VALUE [, VALUE]*`
- `!serious VALUE [, VALUE]*`
- `!macro TITLE [[~]SYMBOL [, [~]SYMBOL]*] { BLOCK }`

- `+TITLE [ARGUMENT [, ARGUMENT]*]`
- `* = EXPRESSION [, MODIFIER]*`
- `!initmem EXPRESSION`
- `!xor EXPRESSION [ { BLOCK } ]`
- `!pseudopc EXPRESSION { BLOCK }`
- `!cpu KEYWORD [ { BLOCK } ]`
- `!al [ { BLOCK } ]`
- `!as [ { BLOCK } ]`
- `!rl [ { BLOCK } ]`
- `!rs [ { BLOCK } ]`
- `!address [ { BLOCK } ]`
- `!cbm`
- `!subzone [TITLE] { BLOCK }`
- `!realpc`

Quick Reference

Source: QuickRef.txt.

This file should give you a basic overview. More specialized stuff like forcing a specific addressing mode is discussed in extra files ("AddrModes.txt" in this case).

Example of what an ACME source code file looks like

```
;--- Example code fragment, start ---
```

```
        !to "tiny.o", cbm   ; set output file and format
        * = $c000       ; set program counter
```

```
    CLEAR = 147     ; a global symbol definition
!addr   basout = $ffd2      ; another one, marked as an address
```

```
        ; a string output loop:
        ldx #0
```

```
        beq +           ; enter loop
```

```
-           jsr basout  ; output character
            inx     ; advance pointer
+           lda .string, x  ; get character
            bne -       ; check whether last
        rts
.string     !pet "Dumb example", 13, 0
```

```
;--- Example code fragment, end ---
```

Here's the same fragment again, now with some additional info:

```
;--- Example code fragment, start ---
```

```
        !to "tiny.o", cbm   ; set output file and format
; This is a pseudo opcode to select the output filename and format.
; This can also be done using the command line options "-o" and "-f",
; respectively.
        * = $c000       ; set program counter
; This can also be done using the command line option "--setpc".
```

```
    ; some global symbol definitions
    CLEAR = 147 ; this is a simple constant
; Now "CLEAR" is defined as a global symbol having the value 147.
!addr   basout = $ffd2  ; this gets marked as an address
; Now "basout" is defined as a global "address" type symbol having the
; value $ffd2.
    ; The distinction between addresses and non-addresses only
    ; matters when the type check system gets activated using
    ; the "-Wtype-mismatch" switch. Then, a line like
```

```
        ; "lda CLEAR" would trigger a type mismatch warning because
        ; of the missing '#' character.
```

```
          ; a string output loop:
          ldx #0
          beq +           ; enter loop
; "+" is an anonymous forward label. Other ones are "++", "+++", etc.
; They can be used like any other symbol, but they always reference
; their *NEXT* definition. This saves having to think of names for
; unimportant labels. As the label's value is not defined yet, ACME
; will need to perform a second pass.
-         jsr basout  ; output character
; "-" is an anonymous backward label. Other ones are "--", "---", etc.
; They can be used like any other symbol, but they always reference
; their *PREVIOUS* definition. This saves having to think of names for
; unimportant labels. In the line above, the value of "-" is set to
; the current program counter.
          inx     ; advance pointer
+         lda .string,x   ; get character
; Here the value of "+" is set to the current program counter.
; ".string" is a local symbol (because its name starts with a '.'
; character), but as its value is not defined yet, ACME will need to
; perform a second pass.
          bne -       ; check whether last
; Here the last definition of the anonymous "-" label is referenced.
          rts
.string   !pet "Dumb example", 13, 0
; Now the value of the local label ".string" is set to the current
; program counter. All label values are defined now, so after having
; done the second pass, the binary will be saved. The "!pet" pseudo
; opcode stores its string argument in PetSCII encoding to memory,
; followed by the given byte values.
```

```
;--- Example code fragment, end ---
```

As you can see, pseudo opcodes are prefixed with an exclamation mark. That's non-standard, but: Backwards compatibility is the root of all evil. :)

Summary about symbols:

There are global symbols (their names starting with a letter or an underscore character). These can be accessed throughout the whole assembly. Then there are local symbols (their names starting with a '.' character). These can only be accessed from inside the macro or zone they were defined in (for more about macros and zones, see the file "AllPOs.txt").

There are also "cheap locals": their names start with an '@'. The area where these can be accessed is limited automatically by the previous and the following global label (cheap locals are "cheap" because you don't have to put in any extra work to limit their range). And then there are anonymous labels (their names being sequences of either '-' or '+' characters). They are also local (bound to their macro/zone), but in addition to that, the "-" labels can only be used for backward references, while the "+" labels can only be used for forward references. In contrast to global and local labels, anonymous labels can not be defined explicitly (as in SYMBOL = VALUE). Each macro call automatically gets its own scope for local symbols.

Save the given example source code to a file called "tiny.a" and start acme by typing

```
acme tiny.a
```

ACME will then parse the file and report any errors. An output file will only be generated if there were no errors and if an output filename has been given.

After assembly, the example program can be run on a C64 using

```
LOAD "tiny.o", 8, 1
SYS 49152
```

Note that ACME does not include any routines for transferring data to a C64. Such tools exist on almost every platform, and I didn't want ACME to become bloatware.

The pseudo opcodes

A list with information on how to use all the Pseudo Opcodes can be found in the file "AllPOs.txt". Here's just a short overview:

```
!byte   !word   !24   !32   !fill   !align
...for directly placing values into the output file.
```

```
!zone   !symbollist
...for defining the scope of local symbols and saving global symbols.
```

```
!convtab   !pet   !raw   !scr   !scrxor   !text
...for converting and outputting strings.
```

```
!do   !endoffile   !for   !if   !ifdef   !ifndef   !set   !while
...for flow control; looping assembly and conditional assembly.
```

```
!binary   !source   !to
...for handling input and output files.
```

```
!pseudopc
...for offset assembly.
```

```
!initmem   *=
...for segment assembly.
```

```
!macro   +
...for defining and calling macros.
```

```
!cpu   !al   !as   !rl   !rs
...for CPU support, especially the 65816 processor.
```

```
!warn   !error   !serious
...for generating warnings, errors and serious errors.
```

```
!address
...to mark symbols as addresses, for the optional type check system.
```

Command line arguments

The command line syntax for calling acme is quite simple:

```
    acme [options] [files]
```

```
Available options are:
    -h, --help             show this help and exit
        This is more or less useless, because the help is also shown
        if ACME is run without any arguments at all.
```

```
    -f, --format FORMAT    set output file format
        Use this with a bogus format type to get a list of all
        supported ones (as of writing: "plain", "cbm" and "apple")
    -o, --outfile FILE     set output file name
        Output file name and format can also be given using the "!to"
        pseudo opcode. If the format is not specified, "!to" defaults
        to "cbm", while the command line option defaults to "plain".
```

```
    -r, --report           set report file name
        This creates a text listing containing the original line
```

```
    number, the resulting memory address, the byte value(s) put
    there and the original text line from the source file.
```

```
-l, --symbollist FILE  set symbol list file name
    This can also be given using the "!symbollist"/"!sl" pseudo
    opcode. The switch was called "--labeldump" in older versions,
    that name still works, too.
```

```
--vicelabels FILE      set file name for label dump in VICE format
    The resulting file uses a format suited for the VICE emulator.
```

```
--setpc NUMBER         set program counter
    This can also be given in the source code using "* = NUMBER".
```

```
--cpu CPU_TYPE         set target processor
    This can be changed in the source code using the "!cpu" pseudo
    opcode. Defaults to 6502.
    Use this with a bogus cpu type to get a list of all supported
    ones.
```

```
--initmem NUMBER       define 'empty' memory
    This can also be given using the "!initmem" pseudo opcode.
    Defaults to zero.
```

```
--maxerrors NUMBER     set number of errors before exiting
    If not given, defaults to 10.
```

```
--maxdepth NUMBER      set recursion depth for macro calls and !src
    The default value for this is 64.
```

```
--ignore-zeroes        do not determine number size by leading zeroes
    Normally, using leading zeroes forces ACME to generate
    oversized addressing modes, like 3-byte absolute instructions
    instead of 2-byte zero page instructions.
    Using this CLI switch disables this behavior.
```

```
--strict-segments      turn segment overlap warnings into errors
    When changing the program counter, segment overlap warnings may
    be generated. Using this CLI switch turns those warnings into
    errors (which is recommended).
    This strict behavior may become the default in future releases!
```

```
-vDIGIT                set verbosity level
    Sets how much additional informational output is generated.
    Higher values mean more output:
    acme -v0 source.a
        This is the default: No additional output is generated,
        ACME will only display warnings and errors.
    acme -v1 source.a
        Now the start and end addresses of the generated output
        file are displayed, along with its size (a CBM-style
        "load address" is *not* counted).
    acme -v2 source.a
        In addition to the "-v1" output, ACME will announce each
        pass, will show amount and offset of "!binary" loads, and
        show start and end addresses and size of each segment.
    acme -v3 source.a
        In addition to the "-v2" output, ACME will now announce
        each source file.
```

```
-DSYMBOL=VALUE         define global symbol
    This option is useful if you build your projects using
    Makefiles: "-DSYSTEM=64" could build the C64 version while
    "-DSYSTEM=128" could build the C128 version of the software
    (using conditional assembly in your source code file).
```

```
-I PATH/TO/DIR         add search path for input files
    This option allows to add a directory to the search list for
    input files. If an input file cannot be found in the current
    working directory, all directories in the search list are
    tried (the first match is used).
```

```
-W                     fine-tune amount and type of warnings
    -Wno-label-indent
        Disables warnings about labels not being in the leftmost
        column.
    -Wno-old-for
        Disables warnings about the old "!for" syntax and at the
        same time enables warnings about the _new_ "!for" syntax.
        Internally, this does exactly the same as what happens
        when the "--dialect 0.94.8" CLI switch is used...
    -Wno-bin-len
        Do not complain about unusual number of digits in a binary
        literal.
    -Wtype-mismatch
        Enables type checking system (warns about wrong types).
```

```
--use-stdout           fix for 'Relaunch64' IDE
    With this option, errors are written to the standard output
    stream instead of to the standard error stream.
```

```
--msvc                 output errors in MS VS format
    This changes the format of the error output to that used by
```

```
    a certain commercial IDE.
```

```
--color                uses ANSI color codes for error output
    If your terminal emulation supports ANSI escape codes, use
    this option to have warnings and errors displayed in color.
```

```
--fullstop             use '.' as pseudo opcode prefix
    This changes the prefix character used to mark pseudo opcodes
    from '!' to '.' (so sources intended for other assemblers can
    be converted with less effort).
```

```
--dialect VERSION      behave like different version
    This CLI switch tells ACME to mimic the behavior of an older
    version. Use this with a bogus version to get a list of all
    supported ones.
```

```
--test                 enable experimental features
    This is for people who want to help test new features before
    they are officially announced.
```

```
-V, --version          show version and exit.
```

Platform-specific versions of ACME might offer more options. Since version 0.89, ACME accepts more than one top-level-filename given on the command line.

The expression parser

ACME has a relatively powerful maths parser. This parser is used whenever ACME expects to read a value. Supported operations include addition, subtraction, multiplication, divisions, comparisons, shifts, negation, boolean operations and

some assembler-specific stuff like extracting the "low byte", the "high byte" or the "bank byte" of a value. Calculations are done using either signed (at least 32-bit) integer arithmetic or floating point arithmetic using the C "double" data type. Symbol values are stored the same way.

This is a list of the value formats currently known by ACME:

```
Examples        Notes
-----------------------------------------------------------------------
128             a decimal value, integer
128.5           a decimal value, floating point
$d011           hexadecimal values are indicated by either a
0xffd2              leading "$" or a leading "0x".
&1701           an octal value, indicated by "&"
%1010           binary values are indicated by either a leading "%"
%....#...           or a leading "0b". In binary values, you can
0b01100110          substitute the characters "0" and "1" by "." and
                    "#" respectively. This way the values are much
                    more readable, especially when building bitmapped
                    objects (like C64 sprites or fonts) in your source
                    code.
'p'             single characters in single quotes are converted to
                    their character code. The actual numeric value
                    depends on the current conversion table chosen
                    using the "!ct" pseudo opcode.
"player 2"      double quotes indicate text strings. See below for
                    more information on single vs. double quotes.
[2, 3, 5, 7]    brackets indicate lists. These are useful to group
[0, [x, y], 9]      data, for example when passing an arbitrary number
                    of arguments to a macro.
poll_joy2       a global symbol
.fail           a local symbol, indicated by leading "."
@loop           a "cheap local", indicated by leading "@"
*               the current program counter. During offset assembly,
                    "*" gives the value of the "Pseudo PC". Just to
                    make sure: The value of the program counter is
                    always the value that was valid at the start of
                    the current statement, so
                        !word *, *, *, *
                    will give the same value four times. I think most
                    assemblers do it this way.
```

In older versions of ACME , 'x' and "x" were the same thing, namely the character code of the letter x using the currently selected encoding table. Since release 0.97, anything in single quotes gives the character code (as before), while anything in double quotes is treated as a string object. To be compatible to those older versions, ACME keeps accepting one-char strings in a lot of places where actually single characters are expected.

This is a list of the operators currently known by ACME:

```
Priority Example      Meaning                       Alias         Note
-------------------------------------------------------------------------
 16      is_number(v) these three functions return 1               *3
 16      is_list(v)       if v is the correct symbol               *3
 16      is_string(v)         type and 0 otherwise                 *3
 16      len(v)       length of list or string                     *2
 16      sin(v)       trigonometric sine function
 16      cos(v)       trigonometric cosine function
 16      tan(v)       trigonometric tangent function
 16      arcsin(v)    inverse of sin()
 16      arccos(v)    inverse of cos()
 16      arctan(v)    inverse of tan()
 16      address(v)   mark as address               addr(v)
 16      int(v)       convert to integer
 16      float(v)     convert to float
 15      &symbol      "unpseudopc" symbol (see docs on "!pseudopc")
 14       v[w]        access v with index w                        *2
 13         !  v      bit-wise complement          NOT
 12      v  ^  w      to the power of
 11         -  v      negate
 10      v  *  w      multiply
 10      v  /  w      divide
 10      v DIV w      integer divide
 10      v  %  w      remainder of DIV             MOD
  9      v  +  w      add                                          *3
  9      v  -  w      subtract
  8      v <<  w      shift left                   ASL, LSL
  8      v >>  w      arithmetic shift right       ASR
  8      v >>> w      logical shift right          LSR
  7         <  v      low byte of
  7         >  v      high byte of
  7         ^  v      bank byte of
  6      v <=  w      lower or equal
  6      v  <  w      lower than
  6      v >=  w      higher or equal
  6      v  >  w      higher than
  5      v !=  w      not equal                    <>,  ><         *3
  4      v  =  w      equal                                        *3
  3      v  &  w      bit-wise AND                 AND
  2                   bit-wise exclusive OR        XOR
  1      v  |  w      bit-wise OR                  OR
```

Notes: "*2" means this operator only works on lists and strings. "*3" means this operator works on all three data types (numbers, lists and strings). All other operators only work on numbers.

```
Operations with higher priority are done first. Of course you can
change this using parentheses.
```

```
Note that though there are operators to extract the "low byte", the
"high byte" and the "bank byte", there is no operator to extract the
fourth byte. If you want to access that, shift it down using ">>>" or
"LSR".
In cases where it's not clear which operator was wanted, ACME takes
the longest possible one:
    v<>w    ...checks for "v not equal w"
    v< >w   ...checks for "v smaller than high byte of w"
So you may have to separate operators with spaces to make sure ACME
does what you want.
The "power-of" operator is right-associative, so a^b^c means a^(b^c).
```

Calculating 0^0 (zero to the power of zero) will give 1. If you don't know why I'm telling you this, ask a mathematician. :)

Almost, but not quite, complete syntax

Every ACME source code file consists of a non-negative number of "lines". The lines have to be separated from each other using CR, LF or CRLF characters.

Every line consists of a non-negative number of "statements" and an optional comment. Statements have to be separated from each other using colon (":") characters, the comment has to be prefixed with a semicolon (";") character or two slashes ("//").

Every statement consists of an optional "label" and an optional "command". These are separated from each other using any number of SPACE or TAB characters. If a label has blanks before it, a warning is issued (to spot typing errors - see Errors.txt for more info).

Every symbol name consists of these characters: "a" to "z", "A" to "Z", "0" to "9", the underscore character "_" and all characters with values beyond 127. The first character must not be a digit though. But it can be '.' or '@', making the symbol a local one. Local symbols beginning with '.' are only valid inside the current zone (marked using the "!zone" pseudo opcode) or the current macro. Local symbols beginning with '@' are only valid between the enclosing global labels (or inside the current macro). Two other possibilities for label names are "all-characters-are-minus" (then it is an anonymous backward label) and "all-characters-are-plus" (then it is an anonymous forward label).

```
Every command is one of the following:
    An assembler mnemonic with an optional argument
    A pseudo opcode, beginning with a "!" character
    An explicit symbol definition (SYMBOL = VALUE)
    A pc definition, beginning with a "*" character
    A macro call, beginning with a "+" character
...and the syntax of those things varies. :)
```

Assembler mnemonics and pseudo opcodes are case insensitive, so whether you write "LDA" or "lda" or "LdA" does not make a difference.

In earlier releases of ACME, arithmetic operators like MOD, XOR, LSL had to be written in UPPER CASE. This is no longer needed.

Symbol names are case sensitive, so "label" and "Label" are two different things.

Complete Directive Reference

Source: AllPOs.txt.

This is a list of all the pseudo opcodes currently implemented. Stuff in square brackets is optional, stuff followed by "*" may be given more than once. This list is not sorted alphabetically, the pseudo opcodes are grouped together according to their usage.

How to insert values

`!8`

Syntax

```
!8 EXPRESSION [, EXPRESSION]*
```

Purpose

Insert 8-bit values.

Parameters

```
EXPRESSION: Any formula the value parser accepts.
```

Aliases

```
"!08", "!by", "!byte"
```

Examples

```
!08 127, symbol, -128   ; output some values
!by 14, $3d, %0110, &304, <*, 'c'
!byte 3 - 4, symbol1 XOR symbol2, 2 ^ tz, (3+4)*7
```

`!16`

Syntax

```
!16 EXPRESSION [, EXPRESSION]*
```

Purpose

Insert 16-bit values in chosen CPU's byte order.

Parameters

```
EXPRESSION: Any formula the value parser accepts.
```

Aliases

```
"!wo", "!word" (and because all currently supported
        CPUs are little-endian, "!le16" is in fact another
        alias)
```

Examples

```
!16 65535, symbol, -32768   ; output some values
!wo 14, $4f35, %100101010010110, &36304, *, 'c'
!word 3000 - 4, a1 AND a2, 2 ^ tz, (3+4)*70, l1 & .j2
```

`!le16`

Syntax

```
!le16 EXPRESSION [, EXPRESSION]*
```

Purpose

Insert 16-bit values in little-endian byte order.

Parameters

```
EXPRESSION: Any formula the value parser accepts.
```

Aliases

```
None (but because all currently supported CPUs are
        little-endian, "!16/!wo/!word" are in fact aliases)
```

Examples

```
!le16 65535, symbol, -32768 ; output some values
!le16 14, $4f35, %100101010010110, &36304, *, 'c'
!le16 3000 - 4, a1 AND a2, 2 ^ tz, (3+4)*70, l1 & .j2
```

`!be16`

Syntax

```
!be16 EXPRESSION [, EXPRESSION]*
```

Purpose

Insert 16-bit values in big-endian byte order.

Parameters

```
EXPRESSION: Any formula the value parser accepts.
```

Aliases

```
None
```

Examples

```
!be16 65535, symbol, -32768 ; output some values
!be16 14, $4f35, %100101010010110, &36304, *, 'c'
!be16 3000 - 4, a1 AND a2, 2 ^ tz, (3+4)*70, l1 & .j2
```

`!24`

Syntax

```
!24 EXPRESSION [, EXPRESSION]*
```

Purpose

Insert 24-bit values in chosen CPU's byte order.

Parameters

```
EXPRESSION: Any formula the value parser accepts.
```

Aliases

```
None (but because all currently supported CPUs are
        little-endian, "!le24" is in fact an alias)
```

Examples

```
!24 16777215, symbol, -8388608, 14, $6a4f35
!24 %10010110100101010010110, &47336304, *, 'c'
!24 300000 - 4, a1 AND a2, 2 ^ tz, (3+4)*70, l1 & .j2
```

`!le24`

Syntax

```
!le24 EXPRESSION [, EXPRESSION]*
```

Purpose

Insert 24-bit values in little-endian byte order.

Parameters

```
EXPRESSION: Any formula the value parser accepts.
```

Aliases

```
None (but because all currently supported CPUs are
        little-endian, "!24" is in fact an alias)
```

Examples

```
!le24 16777215, symbol, -8388608, 14, $6a4f35
!le24 %10010110100101010010110, &47336304, *, 'c'
!le24 300000 - 4, a1 AND a2, 2 ^ tz, (3+4)*70, l1 & .j2
```

`!be24`

Syntax

```
!be24 EXPRESSION [, EXPRESSION]*
```

Purpose

Insert 24-bit values in big-endian byte order.

Parameters

```
EXPRESSION: Any formula the value parser accepts.
```

Aliases

```
None
```

Examples

```
!be24 16777215, symbol, -8388608, 14, $6a4f35
!be24 %100101101001010100010110, &47336304, *, 'c'
!be24 300000 - 4, a1 AND a2, 2 ^ tz, (3+4)*70, 11 & .j2
```

`!32`

Syntax

```
!32 EXPRESSION [, EXPRESSION]*
```

Purpose

Insert 32-bit values in chosen CPU's byte order.

Parameters

```
EXPRESSION: Any formula the value parser accepts.
```

Aliases

```
None (but because all currently supported CPUs are
        little-endian, "!le32" is in fact an alias)
```

Examples

```
!32 $7fffffff, symbol, -$80000000, 14, $46a4f35
!32 %10010110100101010011011010010, &4733630435, *, 'c'
!32 300000 - 4, a AND a2, 2 ^ tz, (3+4)*70, l1 & .j2
```

`!le32`

Syntax

```
!le32 EXPRESSION [, EXPRESSION]*
```

Purpose

Insert 32-bit values in little-endian byte order.

Parameters

```
EXPRESSION: Any formula the value parser accepts.
```

Aliases

```
None (but because all currently supported CPUs are
        little-endian, "!32" is in fact an alias)
```

Examples

```
!le32 $7fffffff, symbol, -$80000000, 14, $46a4f35
!le32 %10010110100101010011011010010, &4733630435, *, 'c'
!le32 300000 - 4, a AND a2, 2 ^ tz, (3+4)*70, l1 & .j2
```

`!be32`

Syntax

```
!be32 EXPRESSION [, EXPRESSION]*
```

Purpose

Insert 32-bit values in big-endian byte order.

Parameters

```
EXPRESSION: Any formula the value parser accepts.
```

Aliases

```
None
```

Examples

```
!be32 $7fffffff, symbol, -$80000000, 14, $46a4f35
!be32 %10010110100101010011011010010, &4733630435, *, 'c'
!be32 300000 - 4, a AND a2, 2 ^ tz, (3+4)*70, l1 & .j2
```

`!hex`

Syntax

```
!hex PAIRS_OF_HEX_DIGITS
```

Purpose

```
Insert byte values with a minimum of additional
        syntax. This pseudo opcode was added for easier
        writing of external source code generator tools.
```

Parameters

```
PAIRS_OF_HEX_DIGITS: Just hexadecimal digits, without
        any "0x" or "$" prefix. Spaces and TABs are allowed,
        but not needed to separate the byte values.
```

Aliases

```
"!h"
```

Examples

```
!h f0 f1 f2 f3 f4 f5 f6 f7  ; insert values 0xf0..0xf7
!h f0f1f2f3 f4f5f6f7        ; insert values 0xf0..0xf7
```

```
!h f0f1f2f3f4f5f6f7     ; insert values 0xf0..0xf7
!h f0f 1f2  ; ERROR: space inside pair!
!h 0x00, $00    ; ERROR: "0x", "," and "$" are forbidden!
!h SOME_SYMBOL  ; ERROR: symbols are forbidden!
!h ABCD ; insert value 0xAB, then 0xCD (CAUTION, big-endian)
```

`!fill`

Syntax

```
!fill AMOUNT [, VALUE]
```

Purpose

Fill amount of memory with value.

Parameters

```
AMOUNT: Any formula the value parser accepts, but it
        must be solvable even in the first pass.
        VALUE: Any formula the value parser accepts. If
        omitted, a default value is used (currently zero).
```

Aliases

```
"!fi"
```

Examples

```
!fi 256, $ff    ; reserve 256 bytes
!fill 2     ; reserve two bytes
```

`!skip`

Syntax

```
!skip AMOUNT
```

Purpose

```
Advance in output buffer without starting a new
        segment.
```

Parameters

```
AMOUNT: Any formula the value parser accepts, but it
        must be solvable even in the first pass (this
        limitation will hopefully be lifted in a future
        release).
```

Aliases

```
None
```

Examples

```
!skip BUFSIZE   ; reserve some bytes
!skip 5     ; reserve five bytes
```

`!align`

Syntax

```
!align ANDVALUE, EQUALVALUE [, FILLVALUE]
```

Purpose

```
Fill memory until a matching address is reached. ACME
        outputs FILLVALUE until "program counter AND ANDVALUE"
        equals EQUALVALUE.
```

Parameters

```
ANDVALUE: Any formula the value parser accepts, but it
        must be solvable even in the first pass.
        EQUALVALUE: Any formula the value parser accepts, but
        it must be solvable even in the first pass.
        FILLVALUE: Any formula the value parser accepts. If it
        is omitted, a default value is used (currently 234,
        that's the opcode of the 6502 CPU's NOP instruction).
```

Examples

```
!align 255, 0   ; align to page (256 bytes)
!align 63, 0    ; align to C64 sprite block (64 bytes)
```

How to insert text strings

`!convtab`

Syntax

```
!convtab KEYWORD [ { BLOCK } ]
or:     !convtab FILENAME [ { BLOCK } ]
```

Purpose

Choose text conversion table.

Parameters

```
KEYWORD: Name of conversion table. Valid names are:
            pet converts to PetSCII
            raw doesn't convert at all
            scr converts to C64 screencode
        FILENAME: File name of conversion table, given in
        "..." quoting (load from current directory) or in
        <...> quoting (load from library). The file must hold
        exactly 256 bytes.
        BLOCK: A block of assembler statements
        Before encountering this PO, ACME defaults to "raw".
        This PO supersedes the now deprecated "!cbm".
```

Aliases

```
"!ct"
```

Examples

```
!convtab raw
!text "Test"    ; outputs $54 $65 $73 $74
!ct pet
!tx "Test"  ; outputs $d4 $45 $53 $54
!ct scr {
    !tx "Test"  ; outputs $54 $05 $13 $14
    !ct "my_own_table_file"
    !tx "abcdefg"   ; whatever... :)
}
!tx "Test"  ; outputs $d4 $45 $53 $54 again
Hint: If you don't want to fiddle with a hex editor to create a
conversion table file, try using ACME:
!to "asc2pet.ct", plain ; no load address
* = 0           ; pc = table index
; first create "as-is" table
!for i, 0, 255 {!byte i}
; now exchange upper and lower case characters
* = 65, overlay
!for i, 1, 26 {!byte * + 128}
```

```
* = 97, overlay
!for i, 1, 26 {!byte * - 32}
The resulting file can be used as a conversion table to convert to
PetSCII (which is useless, because ACME can do so anyway. But you get
the idea).
```

`!text`

Syntax

```
!text STRING_VALUE [, STRING_VALUE]*
```

Purpose

```
Output the given string(s) using the current
        conversion table.
```

Parameters

```
STRING_VALUE: Can be either a string given in double
        quotes or any formula the value parser accepts.
        Please note that formula results won't be converted,
        but single characters involved in calculations will.
```

Aliases

```
"!tx"
```

Examples

```
!text "Loading...", Char_NewLine, "Filename:", 0
```

```
!tx "Offset character is ", offset - 1 + 'a', 0
```

`!pet`

Syntax

```
!pet STRING_VALUE [, STRING_VALUE]*
```

Purpose

```
Output the given string(s) using the PetSCII
        conversion table (This means to exchange the upper-
        and lowercase characters; useful for C64 programs).
```

Parameters

```
STRING_VALUE: Can be either a string given in double
        quotes or any formula the value parser accepts.
        Please note that formula results won't be converted,
        but single characters involved in calculations will.
```

Examples

```
!pet "Loading...", Char_NewLine, "Filename:", 0
!pet "Offset character is ", offset - 1 + 'a', 0
```

`!raw`

Syntax

```
!raw STRING_VALUE [, STRING_VALUE]*
```

Purpose

```
Output the given string(s) without any conversion at
        all.
```

Parameters

```
STRING_VALUE: Can be either a string given in double
        quotes or any formula the value parser accepts.
```

Examples

```
!raw "Loading...", Char_NewLine, "Filename:", 0
!raw "Offset character is ", offset - 1 + 'a', 0
```

`!scr`

Syntax

```
!scr STRING_VALUE [, STRING_VALUE]*
```

Purpose

```
Output the given string(s) using the C64 screen code
        conversion table (useful for C64 programs, as you will
        have guessed).
```

Parameters

```
STRING_VALUE: Can be either a string given in double
        quotes or any formula the value parser accepts.
        Please note that formula results won't be converted,
        but single characters involved in calculations will.
```

Examples

```
!scr "Loading...", Char_NewLine, "Filename:", 0
!scr "Offset character is ", offset - 1 + 'a', 0
```

`!scrxor`

Syntax

```
!scrxor XOR_VALUE, STRING_VALUE [, STRING_VALUE]*
```

Purpose

```
Output the given string(s) using the C64 screen code
        conversion table and exclusive-OR-ing the results with
        the given value (useful for C64 programs when inverse
        video is needed, or EBC mode, etc.).
```

Parameters

```
XOR_VALUE: Any formula the value parser accepts.
        STRING_VALUE: Can be either a string given in double
        quotes or any formula the value parser accepts.
        Please note that formula results will be neither
        converted nor exclusive-OR-d.
        Single characters involved in calculations will be
        converted, but not exclusive-OR-d.
```

Examples

```
!scrxor $80, "Loading..."
!scrxor $a0, "Offset char is ", (offset-1+'a') XOR $a0
```

File stuff

`!to`

Syntax

```
!to FILENAME, FILEFORMAT
```

Purpose

```
Define the output file name and file type. If this
        opcode isn't used, ACME still fully processes the
        source code - as the resulting binary isn't stored,
        this only serves to check for errors. Instead of using
        this pseudo opcode, you can also use the command line
        options "--outfile" and "--format".
```

Parameters

```
FILENAME: A file name given in "..." quoting.
        FILEFORMAT: Name of file format. Valid names are:
            cbm with load address (Commodore format)
            plain   without load address
            apple   with load address and length (Apple II)
        If FILEFORMAT is omitted, ACME gives a warning and
        then defaults to "cbm" (this can be changed using the
        command line option "--format").
```

Examples

```
!to "eprom.p", plain    ; don't add a load address
!to "demo.o", cbm   ; add c64-style load address
```

`!source`

Syntax

```
!source FILENAME
```

Purpose

```
Assemble another source code file. After having
        processed the new file, ACME continues processing the
        old one.
```

Parameters

```
FILENAME: A file name given in "..." quoting (load
        from current directory) or in <...> quoting (load from
        library).
```

Aliases

```
"!src"
```

Examples

```
!source <6502/std.a>    ; Read library file
!src "Macros.a"     ; Read file from current dir
```

`!binary`

Syntax

```
!binary FILENAME [, [SIZE] [, [SKIP]]]
```

Purpose

Insert binary file directly into output file.

Parameters

```
FILENAME: A file name given in "..." quoting (load
        from current directory) or in <...> quoting (load from
        library).
        SIZE: Any formula the value parser accepts, but it
        must be solvable even in the first pass. If SIZE is
        given, it is used: If the file is longer, only SIZE
        bytes are read; if it is shorter, ACME will use
        padding until SIZE is reached. If SIZE is omitted,
        ACME will include the file until EOF.
        SKIP: Any formula the value parser accepts. If SKIP is
        omitted, it defaults to zero. ACME will start loading
        the file from file offset SKIP. So C64 coders wanting
        to include C64 files without their load addresses
        should use a SKIP value of 2.
```

Aliases

```
"!bin"
```

Examples

```
!binary <Own/menudata.b>    ; insert library file
!bin "asc2pet.b", 256, 2    ; insert 256 bytes
                ; from file offset 2.
```

```
!bin "table", 2, 9  ; insert 2 bytes from offset 9
!bin "list",, 9     ; insert from offset 9 to EOF
```

Symbols

`!zone`

Syntax

```
!zone [TITLE] [ { BLOCK } ]
```

Purpose

```
Switch to new zone of local symbols. Zones can either
        be nested or used sequentially.
```

Parameters

```
TITLE: May consist of letters and digits. Its only
        purpose is to be displayed in error messages, so it'll
        be omitted in most cases.
        BLOCK: A block of assembler statements
        If no block is given, the previous zone is terminated
        and the new zone is started.
        If a block is given, the old zone continues after the
        block.
```

Aliases

```
"!zn"
```

Examples

```
.backgroundcolor = 0    ; some local symbol
!zone File_IO       ; new zone begins here, so
.backgroundcolor = 1    ; this is a different symbol
!zn LinkedList_Init
.backgroundcolor = 2
!zone LinkedList {  ; start of nested zone
    ; imagine some code here...
    !zone LinkedList_Init
    ; imagine some more code here...
    !zone LinkedList_Body {
        ; imagine yet some more code here...
        !zone LinkedList_SecondPart
        ; imagine still some more code here...
    }
    !zone LinkedList_End
    ; you know what to imagine here...
}
.backgroundcolor = 3    ; => "Symbol already defined."
```

`!symbollist`

Syntax

```
!symbollist FILENAME
```

Purpose

```
Write a symbol list to the given file after assembly
        is finished. The list will contain all global symbols.
        This table could be loaded during another assembly
        session using the "!source" pseudo opcode.
```

Parameters

```
FILENAME: A file name given in "..." quoting.
```

Aliases

```
"!sl"
```

Examples

```
!sl "Symbols.a" ; produce symbol list after assembly
!sl "global"    ; produce symbol list after assembly
```

Flow control

`!if`

Syntax

```
!if CONDITION { BLOCK } [ else { BLOCK } ]
```

Purpose

```
Conditional assembly. If the given condition is true,
        the matching block of statements will be parsed;
        if no condition is true, the ELSE block (if present)
        will be parsed.
```

Parameters

```
CONDITION: Any formula the value parser accepts, but
        it must be solvable even in the first pass.
        BLOCK: A block of assembler statements.
```

Examples

```
; Choose word according to "country" symbol:
!if country = uk {
    !text "Grey"
} else if country = fr {
    !text "Gris"
} else if country = de {
    !text "Grau"
} else {
    !text "Gray"
}

; Insert debug code depending on symbol "debug":
!if debug { lda #'z':jsr char_output }
```

`!ifdef`

Syntax

```
!ifdef SYMBOL { BLOCK } [ else { BLOCK } ]
or:     !ifdef SYMBOL STATEMENT
```

`!ifndef`

Syntax

```
!ifndef SYMBOL { BLOCK } [ else { BLOCK } ]
or:     !ifndef SYMBOL STATEMENT
```

Purpose

```
Conditional assembly, depending on whether a symbol is
        already defined or not.
        With "ifdef", if the symbol is defined, the first
        block of statements will be parsed; if it isn't, the
        second block will be parsed instead (if present).
        With "ifndef", it's the other way around: If the
        symbol isn't defined, the first block of statements
        will be parsed; if it is defined, the second block
```

```
        will be parsed instead (if present).
        CAUTION: These opcodes were added to speed up parsing
        of library files (see example below). They can be used
        to tell passes apart, therefore only use them in your
        own files if you're sure you *really* know what you
        are doing - using them in the wrong way will result in
        loads of error messages.
```

Parameters

```
SYMBOL: Any valid symbol name.
        BLOCK: A block of assembler statements.
        STATEMENT: Any assembler statement.
```

Examples

```
; this was taken from <6502/std.a>:
!ifdef Lib_6502_std_a !eof  ; in later passes,
Lib_6502_std_a = 1      ; skip this file.
; During the first pass, the symbol is not defined,
; therefore the file will get parsed. During all
; further passes, the symbol is already defined,
; therefore the file will be skipped.

; if the following code gets included several times,
; only assemble it at the first location:
!ifndef my_label {my_label} ; only define if undefined
!if * = my_label {
    ; imagine some code here...
    ; this block will only be assembled at the
    ; first location where it is included. all
    ; further instances will be skipped.
}

; include at most one driver source code:
!ifdef RAM_REU {
    !src "driver_reu.a"
} else ifdef RAM_GEORAM {
    !src "driver_georam.a"
} else ifdef RAM_VDCRAM {
    !src "driver_vdcram.a"
} else ifdef RAM_SUPERRAM {
    !src "driver_superram.a"
} else {
```

```
        !src "driver_noram.a"
    }
```

`!for`

Syntax

```
    !for SYMBOL, START, END { BLOCK }
```

Purpose

```
    Looping assembly. The block of statements will be
            parsed a fixed number of times, as specified by the
            values of START and END. For more flexible
            possibilities, have a look at "!do" and "!while"
            below.
```

Parameters

```
    SYMBOL: Any valid symbol name.
            START: Any formula the value parser accepts, but it
            must be solvable even in the first pass. SYMBOL will
            have this value during the first loop cycle.
            END: Any formula the value parser accepts, but it must
            be solvable even in the first pass. SYMBOL will have
            this value during the last loop cycle.
            BLOCK: A block of assembler statements.
            If START or END are floats, they will be converted to
            integers (never use floats for loop counters). If
            START is less than or equal to END, SYMBOL will get
            incremented at the end of each cycle; if START is
            greater than END, SYMBOL will get decremented at the
            end of each cycle. So after leaving the loop, SYMBOL
            will have an "illegal" value (END + 1 if counting up,
            END - 1 if counting down).
            Please note that it is impossible to change the number
            of loop cycles "inside" the loop by fiddling with the
            counter using the "!set" pseudo opcode: The "!for"
            routine keeps its own copy of the counter value and
```

```
        only sets the symbol value, it never reads it back.
        This was done to eliminate a possibility to hang ACME.
```

Examples

```
    int2BCD ; conversion table: integer to BCD
!for Outer, 0, 9 {
    !for Inner, 0, 9 {
        !byte (Outer << 4) OR Inner
    }
}
!fill 156, $ff  ; values above 99 give 255 (invalid)

    BCD2int ; conversion table: BCD to integer
!for Outer, 0, 9 {
    !for Inner, 0, 9 {
        !byte 10 * Outer + Inner
    }
    !fill 6, $ff    ; invalid BCD values give 255
}
!fill 96, $ff       ; invalid BCD values give 255

    quickclear ; generate speedcode to clear C64 screen
lda #' '
!for i, 0, 999 {
    sta $0400 + i
}

Miscellaneous:  The old syntax ("!for SYMBOL, END { BLOCK }" where
START was always implied to be 1) is still fully
supported, but gives a warning to get people to change
to the new syntax.
You can disable this warning using the "--dialect" or
the "-Wno-old-for" switches, but then you will get
warnings for using the *new* syntax.
When migrating your sources to the current syntax,
bear in mind that it is no longer possible to skip the
block completely by specifying a loop count of zero.
Also note that with the new algorithm, SYMBOL has a
different value after the block than during the last
loop cycle, while the old algorithm kept that last
value.
```

`!set`

Syntax

```
!set SYMBOL = VALUE
```

Purpose

```
Assign given value to symbol even if the symbol
        already has a different value. Needed for loop
        counters when using "!do"or "!while", for example.
        Only use this opcode for something else if you're sure
        you *really* know what you are doing... :)
```

Parameters

```
SYMBOL: Any valid symbol name.
        VALUE: Any formula the value parser accepts.
```

Example

```
see "!do" below
```

`!do`

Syntax

```
!do [KEYWORD CONDITION] { BLOCK } [KEYWORD CONDITION]
```

Purpose

```
Looping assembly. The block of statements can be
        parsed several times, depending on the given
        condition(s).
```

```
        Conditions may be placed before or after the block (or
        even at both places), they are then parsed in every
        repetition before or after the block respectively. If
        there is a condition before the block and it isn't
        met when first checked, the block will be skipped.
```

Parameters

```
KEYWORD: Either "until" or "while" (without quotes).
        CONDITION: Any formula the value parser accepts, but
        it must be solvable even in the first pass.
        BLOCK: A block of assembler statements.
```

Examples

```
; a loop with conditions at both start and end
!set a = 0          ; init loop counter
!do while loop_flag = TRUE {
    lda #a
    sta label + a
    !set a = a + 1
} until a > 6

; a loop with a condition at the start
!do while * < $c000 { nop }

; a loop with a condition at the end
!do { !wo * + base } while * < base + 345

; a never ending loop - this will cause an error
!do while 3 < 4 { nop } until 3 = 4

; an empty loop - this will hang ACME
!do until 3 = 4 {     } while 3 < 4
```

`!while`

Syntax

```
!while [CONDITION] { BLOCK }
```

Purpose

```
Looping assembly. The block of statements can be
        parsed several times, depending on the given
        condition.
        The condition is parsed in every repetition before the
        actual block. If it isn't met when first checked, the
        block will be skipped.
```

Parameters

```
CONDITION: Any formula the value parser accepts, but
        it must be solvable even in the first pass.
        BLOCK: A block of assembler statements.
```

Examples

```
; a loop with a counter
!set a = 0          ; init loop counter
!while a < 6 {
    lda #a
    sta label + a
    !set a = a + 1
}

; a loop depending on program counter
!while * < $c000 { nop }

; a never ending loop - this will cause an error
!while 3 < 4 { nop }

; an empty loop - this will hang ACME
!while 3 != 4 {     }
```

`!endoffile`

Syntax

```
!endoffile
```

Purpose

```
Stop processing the current source file. Using this
        pseudo opcode you can add explanatory text inside your
        source file without having to comment out every single
        line of it.
```

Aliases

```
"!eof"
```

Example

```
rts ; some assembler mnemonic
!eof
Though this text isn't preceded by a semicolon, it is
treated as if it were a comment. In fact, ACME doesn't
even parse this anymore - the file gets closed when
"!eof" is reached.
```

`!warn`

Syntax

```
!warn VALUE [, VALUE]*
```

Purpose

Show a warning during assembly.

Parameters

```
VALUE: Can be either a string given in double quotes
        or any formula the value parser accepts.
        Integer numbers will be output in both decimal _and_
        hex formats.
```

Example

```
!if * > $a000 {
    !warn "Program reached ROM: ", * - $a000, " bytes overlap."
}
```

`!error`

Syntax

```
!error VALUE [, VALUE]*
```

Purpose

```
Generate an error during assembly (therefore, no
        output file will be generated).
```

Parameters

```
VALUE: Can be either a string given in double quotes
        or any formula the value parser accepts.
        Integer numbers will be output in both decimal _and_
        hex formats.
```

Example

```
rts ; end of some function
    start   !source "colors.a"
    end !if end - start > 256 {
    !error "Color strings are ", end - start - 256, " bytes too long."
}
```

`!serious`

Syntax

```
!serious VALUE [, VALUE]*
```

Purpose

```
Generate a serious error, immediately stopping
        assembly.
```

Parameters

```
VALUE: Can be either a string given in double quotes
        or any formula the value parser accepts.
        Integer numbers will be output in both decimal _and_
        hex formats.
```

Example

```
!source "part1.a"   ; sets part1_version
!source "part2.a"   ; sets part2_version
!if part1_version != part2_version {
    !serious "part1.a and part2.a don't match!"
}
```

🧩 Macro Reference

> **Macro quick reference**
> ACME macros are defined with `!macro` and invoked with `+NAME`. Each invocation gets its own scope for local symbols. The two entries below are the core macro features; the complete macro library is documented later in this file.

`!macro` — Define a Macro

Syntax

Syntax

```
!macro TITLE [[~]SYMBOL [, [~]SYMBOL]*] { BLOCK }
```

Purpose

Define a macro.

Parameters

```
TITLE: The macro's desired name (same rules as for
        symbol names). If the title's first character is a dot
        ("."), the macro will be local (though why anyone
        could want this is beyond me).
        SYMBOL: The desired name for the parameter value at
        call time. Normally, these parameter symbols should be
        local (first character a '.' or a '@'), as different
        macro calls will almost for sure have different
        parameter values.
        If you prefix SYMBOL with a '~' character, it will be
        called by reference, not by value: Changing the value
        inside the macro will result in the "outer" symbol to
        be changed as well.
        BLOCK: A block of assembler statements.
```

Examples

```
; far branch, as defined in <6502/std.a>
!macro bne .target {
    beq * + 5
    jmp .target
}
```

```
; increase 16-bit counters
!macro dinc .target {
    inc .target
    bne +   ; "bne * + 5" would not work in zp
    inc .target + 1
+
}
; Yes, anonymous label references can be used with
; macros (unlike several other assemblers). That's
; because ACME's macros are implemented more like
; real functions.

; load A and X
!macro ldax .target {
    lda .target
    ldx .target + 1
}

; store A and X
!macro stax .target {
    sta .target
    stx .target + 1
}

; use call-by-reference for return value
!macro reserve ~.address, .amount {
    .address = external_pc
    !set external_pc = external_pc + .amount
}

; define a pixel row of a C64 hardware sprite
!macro SpriteLine .v {
    !by .v >> 16, (.v >> 8) & 255, .v & 255
}
```

`+TITLE`

Syntax

```
+TITLE [ARGUMENT [, ARGUMENT]*]
```

Purpose

Call a macro, using the given parameter values.

Parameters

```
TITLE: The macro's name as given in its definition.
        ARGUMENT: This is either any formula the value parser
        accepts, or (new in release 0.86) a '~' character
        followed by a symbol name. The '~'-prefix indicates
        call-by-reference semantics, which means that when the
        macro changes the symbol's value, the caller's
        symbol's value will change as well.
```

Examples

```
inc label
bne mark    ; "near" branch
inc label2
+bne mark2  ; "far" branch

inc $fa     ; increase  8-bit counter
+dinc $fb   ; increase 16-bit counter

ldy label   ; get byte
+ldax label2    ; get two bytes

; using macro calls in a macro definition
!macro cp16 .source, .target {
    +ldax .source
    +stax .target
}

; use call-by-reference for return value
!set external_pc = $0400
+reserve ~.line_buffer, 80
+reserve ~.in_buffer, 256
+reserve ~.out_buffer, 256
+reserve ~.byte_var, 1

; define a C64 hardware sprite
;            76543210765432107654321 0
+SpriteLine %........................
+SpriteLine %.#......................
+SpriteLine %.##.....................
+SpriteLine %.###....................
+SpriteLine %.####...................
+SpriteLine %.#####..................
+SpriteLine %.######.................
+SpriteLine %.#######................
+SpriteLine %.########...............
+SpriteLine %.#########..............
+SpriteLine %.########...............
```

```
+SpriteLine %.######.................
+SpriteLine %.######.................
+SpriteLine %.##..##.................
+SpriteLine %.#....##................
+SpriteLine %......##................
+SpriteLine %.......##...............
+SpriteLine %.......##...............
+SpriteLine %........##..............
+SpriteLine %........##..............
+SpriteLine %........................
!byte 0 ; pad to 64-byte block

Since release 0.86, different macros are allowed to have the same name
as long as their parameter lists differ in size (number of arguments)
or type (call-by-value vs. call-by-reference). So
!macro process_bytes b1, b2 {...whatever...}
!macro process_bytes b1, b2, b3 {...whatever...}
!macro process_bytes b1, b2, ~b3 {...whatever...}
can *all* be used at the same time without any name clash.
```

Segment assembly

`*`

Syntax

```
* = EXPRESSION [, MODIFIER]*
```

Purpose

```
Set program counter to given value and start new
        segment. This opcode must be given at least once
        (or the command line option "--setpc" must be used).
        If segments overlap each other, warnings will be
        issued. Because some people do this overlapping
        on purpose, the warnings can be suppressed using
        modifier keywords.
        Using the "--strict-segments" CLI switch, these
        warnings can be turned onto errors. Future versions of
        ACME may do that by default - so if needed, use the
        modifier keywords.
```

Parameters

```
EXPRESSION: Any formula the value parser accepts, but
        it must be solvable even in the first pass.
        MODIFIER: "overlay" or "invisible" (without quotes):
        "overlay" suppresses the warning "Segment starts
        inside another one, overwriting it".
        "invisible" makes the new segment invisible, so that
        _other_ segments will never raise the warning "Segment
        reached another one, overwriting it".
```

Examples

```
!to "TinyDemo", cbm ; define output file + format
* = $0801       ; start at C64 BASIC start
!src "basicmacros.a"    ; include macro definitions
+basic_header       ; call program header macro
!src "main.a"       ; include main program
* = $1000       ; jump to new segment
!bin "music.b"      ; load music to $1000
* = $8000       ; jump to new segment
!bin "pic.b"        ; load graphics to $8000
* = $8010, overlay, invisible   ; go back and patch
        ; the graphics, suppressing warnings
; After assembly, ACME will save everything from $0801
; up to the highest address written to. The resulting
; file will contain some big unused areas (zero'd),
; but demos will get compressed anyway... :)
```

`!initmem`

Syntax

```
!initmem EXPRESSION
```

Purpose

```
Define "unchanged" memory. ACME will fill its output
        buffer completely with the given value before storing
        the assembled code. So gaps between segments will
        contain the desired byte when writing the output file.
        Instead of using this pseudo opcode, you can also use
        the "--initmem" command line option. If neither is
        used, the buffer is cleared.
```

Parameters

```
EXPRESSION: Any formula the value parser accepts, but
        it must be solvable even in the first pass (because
        this opcode will be ignored in all later passes).
```

Examples

```
!to "TinyDemo", cbm ; define output file + format
!initmem $ea        ; default memory content $ea.
* = $0801       ; start at C64 BASIC start
!src "basicmacros.a"    ; include macro definitions
+basic_header       ; call program header macro
!src "main.a"       ; include main program
* = $1000       ; jump to new segment
!bin "music.b"      ; load music to $1000
* = $8000       ; jump to new segment
!bin "pic.b"        ; load graphics to $8000
* = $8010, overlay, invisible   ; go back and patch
        ; the graphics, suppressing warnings
; This is the same example as before, but now the big
; unused areas will contain the value $ea instead of
; zero.

!initmem $ff    ; Default memory content is now $ff.
; Useful if you want to store your code in an EPROM.
```

`!xor`

Syntax

```
!xor EXPRESSION [ { BLOCK } ]
```

Purpose

```
Change the value to XOR all output bytes with (the
        value defaults to zero on startup). This "encryption"
        facility was added to compensate for the shortcomings
        of the "!scrxor" pseudo opcode, which only XORs
        strings and characters, but not numbers.
        When used with block syntax, the previously chosen
        value is restored afterwards.
```

Parameters

```
EXPRESSION: Any formula the value parser accepts.
        BLOCK: A block of assembler statements.
```

Examples

```
; first as normal screencodes:
!scr "Hello everybody...", GROUPLOGOCHAR
; and now as inverted screencodes:
!xor $80 {
!scr "Hello everybody...", GROUPLOGOCHAR
}
```

Offset assembly

`!pseudopc`

Syntax

```
!pseudopc EXPRESSION { BLOCK }
```

Purpose

```
Assemble code as if the program counter had the given
        value, effectively producing a program that has to be
        copied to a different address space before being run.
        After having processed the block of statements with
        the new program counter, the updated (!) old program
        counter is used again.
        Thanks to the block syntax, offset assembly can now be
        nested. Then the old program counter would not
        necessarily be the *real* program counter, but could
        be a pseudopc as well. ;)
```

Parameters

```
EXPRESSION: Any formula the value parser accepts, but
        it must be solvable even in the first pass.
        BLOCK: A block of assembler statements.
```

Examples

```
ldx #.shifted_end - .shifted_start
    -       lda .shifted_start - 1, x
    sta .target - 1, x
    dex
    bne -
jmp .target
    .shifted_start
!pseudopc $0400 {
    .target ; imagine some code here...
; it should be copied to $0400 and executed *there*
}
    .shifted_end

Miscellaneous:  If you need to convert a label or the program counter
from its "pseudopc" to its "real" value, you can do
that using the "&" operator. Given the example above,
the symbol ".target" will evaluate to the value $0400,
but "&.target" will evaluate to the same value as
".shifted_start" will.
```

CPU support pseudo opcodes (especially 65816 support)

`!cpu`

Syntax

```
!cpu KEYWORD [ { BLOCK } ]
```

Purpose

```
Select the processor to produce code for. If this PO
        isn't used, ACME defaults to the 6502 CPU (or to the
        one selected by the "--cpu" command line option).
        ACME will give errors if you try to assemble
        instructions the chosen CPU does not support. You can
        change the chosen CPU at any time. When used with
        block syntax, the previously chosen CPU value is
        restored afterwards.
```

Parameters

```
KEYWORD: Currently valid keywords are:
        6502        for the original MOS 6502
        nmos6502    6502 plus undocumented opcodes
        6510        (alias for "nmos6502")
        65c02       6502 plus BRA,PHX/Y,PLX/Y,STZ,TRB/TSB
        r65c02      65c02 plus BBRx, BBSx, RMBx, SMBx
        w65c02      r65c02 plus STP/WAI
        65816       65c02 plus 16/24-bit extensions
        65ce02      r65c02 plus Z reg, long branches, ...
        4502        65ce02 with MAP instead of AUG
        m65     4502 plus 32-bit extensions
        c64dtv2     6502 plus BRA/SAC/SIR plus some of the
                undocumented opcodes
            See "docs/cputypes/all.txt" for more info.
        BLOCK: A block of assembler statements.
```

Examples

```
!if cputype = $65c02 {
    !cpu 65c02 {    ; temporarily allow 65c02 stuff
        stz .todelete
    }
} else {
    pha
    lda #0
    sta .todelete
    pla
}
rts
!cpu 65816  ; now allow instructions of 65816 cpu
```

`!al`

Syntax

```
!al [ { BLOCK } ]
```

Purpose

```
Assume long (16 bits) accumulator. Only allowed when
        producing code for the 65816 CPU. When used with block
        syntax, the previous configuration is restored
        afterwards.
```

`!as`

Syntax

```
!as [ { BLOCK } ]
```

Purpose

```
Assume short (8 bits) accumulator. Only needed when
        producing code for the 65816 CPU. When used with block
        syntax, the previous configuration is restored
        afterwards. Short accumulator is the default in every
        pass.
```

`!rl`

Syntax

```
!rl [ { BLOCK } ]
```

Purpose

```
Assume long (16 bits) index registers. Only allowed
        when producing code for the 65816 CPU. When used with
        block syntax, the previous configuration is restored
        afterwards.
```

`!rs`

Syntax

```
!rs [ { BLOCK } ]
```

Purpose

```
Assume short (8 bits) index registers. Only needed
        when producing code for the 65816 CPU. When used with
        block syntax, the previous configuration is restored
        afterwards. Short registers are the default in every
        pass.
```

Type system

`!address`

Syntax

```
!address [ { BLOCK } ]
or:     !address SYMBOL = VALUE
```

Purpose

```
Mark a block or a statement as "explicitly defined
        symbols are holding addresses".
```

Parameters

```
BLOCK: A block of assembler statements
        Everything inside the block will be parsed as usual,
        but explicitly defined symbols will be marked as
        referencing memory.
        If no block is given, only the current statement will
        be affected, which should then be an explicit symbol
        definition.
        To make use of this feature, you need to use the
        "-Wtype-mismatch" CLI switch.
```

Aliases

```
"!addr"
```

Examples

```
!addr   k_chrout = $ffd2    ; this is an address
    CLEAR = 147     ; but this is not
!addr {
    ; these are addresses:
    sid_v1_control  = $d404
    sid_v2_control  = $d40b
    sid_v3_control  = $d412
}
; these are not:
sid_VOICECONTROL_NOISE      = %#.......
sid_VOICECONTROL_RECTANGLE  = %.#......
sid_VOICECONTROL_SAWTOOTH   = %..#.....
sid_VOICECONTROL_TRIANGLE   = %...#....
```

Obsolete pseudo opcodes (they will throw errors if used)

`!cbm`

Syntax

```
!cbm
```

Purpose

```
Use PetSCII as the text conversion table. Now
        superseded by the "!convtab" pseudo opcode.
Old usage:  !cbm        ; gives "use !ct pet instead" error
Now use:    !convtab pet    ; does the same without error
        If you just want to assemble an old source code
        without touching it, use the "--dialect" CLI switch:
        Using "--dialect 0.94.6" or earlier will assemble this
        pseudo opcode without throwing an error.
```

`!subzone`

Syntax

```
!subzone [TITLE] { BLOCK }
```

Purpose

```
Allows nesting of zones. Now superseded by "!zone"
        because that allows nesting as well.
```

Parameters

```
TITLE: May consist of letters and digits.
        BLOCK: A block of assembler statements.
```

Aliases

```
"!sz"
Old usage:  !subzone graphics {
            !source "graphics.a"
        }
Now use:    !zone graphics {
            !source "graphics.a"
        }
        If you just want to assemble an old source code
        without touching it, use the "--dialect" CLI switch:
        Using "--dialect 0.94.6" or earlier will assemble this
        pseudo opcode without throwing an error.
```

`!realpc`

Syntax

```
!realpc
```

Purpose

```
Restore the program counter to its real value,
        therefore finishing offset assembly. Because
        "!pseudopc" now knows block syntax and can be nested,
        there's no reason to use "!realpc" any more.
Old usage:  !pseudopc $0400
        ; imagine some code here...
        !realpc
Now use:    !pseudopc $0400 {
            ; imagine some code here...
        }
        If you just want to assemble an old source code
        without touching it, use the "--dialect" CLI switch:
        Using "--dialect 0.94.6" or earlier will assemble this
        pseudo opcode without throwing an error.
        Using "--dialect 0.85", not even a warning is thrown.
```

Addressing Modes and Size Forcing

Source: AddrModes.txt.

If an instruction can be used with different addressing modes, ACME has to decide which one to use. Several instructions of the 6502 CPU can be used with either "absolute" addressing or "zeropage-absolute" addressing. The former one means there's a 16-bit argument, the latter one means there's an 8-bit argument. And the 65816 CPU even has some instructions with 24-bit addressing...

So how does ACME know which addressing mode to use? The simple approach is to always use the smallest possible argument, of course: If the argument fits in a byte, use zeropage addressing. If it doesn't, use absolute addressing. If it needs more than two bytes and the 65816 CPU is chosen, use 24-bit addressing.

In most cases this works - with two exceptions. The remainder of this text may sound somewhat confusing now, so if you don't have any problems with addressing modes, then don't bother trying to understand everything this texts says. :)

The two exceptions are:

*** 1) Symbols are defined too late

If ACME cannot figure out the argument value in the first pass, it assumes that the instruction uses 16-bit addressing.

If it later finds out that the argument only needs 8 bits, ACME gives a warning ("using oversized addressing mode") and continues. However, if it finds out that the argument needs 24 bits, it gives an error.

These problems can be solved by defining the symbols *before* using them, so that the value can be figured out in the first pass. If this is not possible, you can use the postfix method, effectively exactly defining what addressing mode to use. The postfix method is described in a separate section below.

*** 2) You *want* to use an oversized addressing mode

On the 65816 CPU, "zeropage addressing" is called "direct page addressing". The difference is that the position of the "direct page" can be changed. Then, "lda $fa" does not necessarily access the same memory location as "lda $00fa" anymore. The same goes for 16- and 24- bit addressing: "lda $fabc" does not necessarily access the same memory location as "lda $00fabc", because the default bank can be set to something other than zero. But even on the plain 6502 CPU you might want to force ACME to use an oversized addressing mode, for example because of timing issues.

Again there are two ways to solve the problem: You can define the target location using leading zeros. ACME will then use an addressing mode that is big enough even if the leading zeros would have been other digits:

```
symbol1 = $fb
symbol2 = $00fd
symbol3 = $0000ff
```

```
    lda $fa
    sta $00fc
    lda $0000fe
    sta symbol1
    lda symbol2
    sta symbol3
```

will be assembled to

```
a5 fa     ; lda $fa
8d fc 00  ; sta $00fc
af fe 00 00   ; lda $0000fe
85 fb     ; sta $fb
ad fd 00  ; lda $00fd
8f ff 00 00   ; sta $0000ff
```

This feature can be disabled using the "--ignore-zeroes" CLI switch.

The other possibility is to use the postfix method (described in the next section).

*** The postfix method

Warning: This may sound very complicated at first, but I think that once you get used to it you'll agree it's useful. If you don't want to use this, stick to the "leading zeros" method and don't bother about postfixes.

Still with me? Okay: You can force ACME to use a specific addressing mode by adding "+1", "+2" or "+3" to the assembler mnemonic. Each one of these postfixes sets the relevant "Force Bit" in ACME's result. If Force Bit 3 is set, ACME will use

24-bit addressing. Force Bit 2 means 16-bit addressing and Force Bit 1 means 8-bit addressing. Higher Force Bits have higher priorities.

Here's an (overly complicated) example:

```
symbol1   = $fb
symbol2   = $fd
symbol3+3 = $ff ; set Force Bit 3 and store in symbol's flags
```

```
    ldx   $fa
    ldy+2 $fc   ; set Force Bit 2 (16-bit addressing)
    lda+3 $fe   ; set Force Bit 3 (24-bit addressing)
    stx   symbol1
    sty+2 symbol2   ; set Force Bit 2 (16-bit addressing)
    sta   symbol3   ; no need to set Force Bit 3 as it is
            ; already set in "symbol3".
```

will be assembled to

```
a6 fa     ; ldx $fa
ac fc 00  ; ldy $00fc
af fe 00 00   ; lda $0000fe
86 fb     ; stx $fb
8c fd 00  ; sty $00fd
8f ff 00 00   ; sta $0000ff
```

Postfixes added directly to the mnemonic have higher priority than those given to the argument. As you can see, you can add the postfix to the symbol definition as well (equivalent to leading zeros).

Applying the byte extraction operators ("<" gives the low byte, ">" gives the high byte and "^" gives the bank byte of a value) to any value will clear the argument's Force Bits 2 and 3 and set Force Bit 1 instead. So "lda <symbol" will use 8-bit addressing, regardless of the symbol's Force Bits. Of course, you can change this by postfixing the instruction again... :)

*** The algorithm to find the right addressing mode

You don't need to read this paragraph just to use ACME, I only included it for completeness' sake. This is a description of ACME's strategy for finding the addressing mode to use:

First, ACME checks whether the instruction has any postfix. If it has, ACME acts upon it. So postfixes have the highest priority.

Otherwise, ACME checks whether the argument has any Force Bits set (because of leading zeros or byte extraction operators, for example) or references symbols that have. If any Force Bit is set, ACME acts upon it. This is the next priority level: Leading zeros or postfixes added to the symbol definition.

Otherwise, ACME checks whether the argument could be figured out in the first pass. If it couldn't, ACME tries to use a default addressing mode (normally this will be 16-bit addressing). In case the value could be figured out even in the first pass, then everything's cool and froody: ACME just looks at the argument's value and uses the smallest addressing mode that matches.

I admit that this algorithm sounds complicated, but it hasn't failed yet. :) And in everyday usage, it does exactly what one expects.

If you want to take a closer look at the algorithm, examine the "calc_arg_size" function in the "src/mnemo.c" file.

Errors and Solutions

Source: Errors.txt.

Here's a sorted list of all error messages ACME can give, possible reasons and what you can do to sort it out.

Errors on startup

```
Cannot open toplevel file "FILENAME".
    Maybe you mistyped its name?
```

```
Error in CLI arguments: ...
    There are several of these errors, but they should be quite self-
    explanatory.
```

Warnings during assembly

```
!warn: ...
    This is given when the pseudo opcode "!warn" is executed. The
    actual message varies according to the pseudo opcode's arguments.
```

```
Assembling buggy JMP($xxff) instruction
    The original 6502 processor has a bug: When executing an indirect
    JMP instruction where the low byte of the argument equals $ff, it
    fetches the high byte of the jump target address not from memory
    location ARGUMENT + 1, but from ARGUMENT - 255. Therefore ACME
    issues this warning if you are about to generate such an
    instruction.
    Note that this warning is only given for some CPU types (6502,
    nmos6502/6510, c64dtv2) because later ones like 65c02 and 65816
    have been fixed in this regard.
```

```
Assembling unstable ANE #NONZERO instruction
Assembling unstable LXA #NONZERO instruction
    These warnings are only ever given for CPU type nmos6502 (6510).
    ANE and LXA are undocumented ("illegal") opcodes of this CPU, and
    they only work reliably if the argument is zero or the accumulator
    contains 0xff.
    Therefore ACME issues these warnings if it is about to generate
    these instructions with a non-zero argument.
```

```
Binary literal without any digits.
Hex literal without any digits.
    A special literal was started, but then no digits followed. Expect
    this to become an error in future!
```

```
Binary literal with strange number of digits.
    This warning is given if the number of digits in a binary literal
    is not a multiple of four. This is useful when you meant to write
    %#....... but actually wrote %#........ by mistake. See? :P
    You can disable this warning using the CLI switch "-Wno-bin-len".
```

```
Bug in ACME, code follows
    A situation has been encountered implying there is a bug in ACME.
```

```
    See the last section in this file.
```

```
C-style "==" comparison detected.
    To check for equality, use a single '=' character instead.
```

```
Converted to integer for binary logic operator.
    Applying binary logic to float values does not make much sense,
    therefore floats will be converted to integer in such cases.
```

```
"EOR" is deprecated; use "XOR" instead.
    This means the operator, not the mnemonic.
```

```
Found old "!for" syntax.
    Please update your sources to use the new "!for" syntax. See
    AllPOs.txt for details.
    You can suppress this warning using the "--dialect" or the
    "-Wno-old-for" CLI switch.
    ("-Wno-old-for" does _exactly_ the same as "--dialect 0.94.8")
```

```
Found new "!for" syntax.
    When using the "-Wno-old-for" switch to disable the warning about
    the older syntax, the new syntax will trigger this warning.
```

```
Found SED instruction for CPU with known decimal SBC bug.
    This warning is only ever given for CPU types 65ce02 and 4502,
    because they are known to be buggy in decimal mode.
```

```
    Pavel Zima and Eric Smith found an example where $41 minus $08
    gave $39 instead of $33.
```

```
Label name not in leftmost column.
    A label definition has blanks before the label name.
    Imagine this source code:
        lda #00
        imx
        rts
    Obviously, there's a typo in the middle line (imx instead of inx),
    but ACME does not recognize this: It looks just like a label
    definition! Therefore releases 0.89 and higher warn you when a
    label name does not start in column 1. Releases 0.94 and higher
    support a command line option to switch off this warning
    ("-Wno-label-indent").
```

```
Label name starts with a shift-space character.
    The name of a global label starts with a shift-space character. It
    is highly likely that this is a typing error, therefore this
    warning is issued.
```

```
Memory already initialised.
    The "!initmem" command was given more than once (or in addition to
    the "--initmem" command line option). Only use it once.
```

```
Output file already chosen.
    The "!to" command was given more than once (or in addition to the
    "--outfile" command line option). Only use it once.
```

```
Segment reached another one, overwriting it.
    The program counter has just reached the start of another segment.
    Because some people might want to assemble "onto" a binary file
```

```
    that was loaded before, this warning can be inhibited using
    modifier keywords when changing the program counter via "*=".
    For extra safety you can also turn this warning into an error
    using the "--strict-segments" CLI switch. In future versions of
    ACME this might become the default.
```

```
Segment starts inside another one, overwriting it.
    The given value in a "*=" command is located inside another
    segment. Because some people might want to assemble "onto" a
    binary file that was loaded before, this warning can be inhibited
    using modifier keywords when changing the program counter via
    "*=".
    For extra safety you can also turn this warning into an error
    using the "--strict-segments" CLI switch. In future versions of
    ACME this might become the default.
```

```
Symbol list file name already chosen.
    The "!sl" command was given more than once (or in addition to the
    "--symbollist" command line option). Only use it once.
```

```
Used "!to" without file format indicator. Defaulting to "cbm".
    Now that "!to" can be given a file format keyword (either "plain"
    or "cbm"), using "cbm" as default seems inappropriate. It still
    works though.
```

```
Using oversized addressing mode.
    ACME just assembled a command using an addressing mode that was
    larger than needed. This only happens if ACME could not work out
    the argument's value in the first pass, therefore assuming a 16-
    bit addressing mode. If, in a later pass, ACME finds out that the
    argument is small enough to fit in 8 bits, then this warning is
    shown. If you define all your zeropage symbols *before* they are
    first used, this shouldn't happen. If you know that a specific
    argument fits in 8 bits, you can force ACME to use 8 bits
    addressing by postfixing the command with "+1". Example:
        lda+1 label
```

```
    ACME will then use an 8-bit addressing mode, regardless of whether
    the label is known or not. If the label value happens to be too
    large to fit in 8 bits, ACME will show an error of course (To
    always truncate a value to 8 bits, use the '<' operator).
    More about the postfixing method can be found in "AddrModes.txt".
```

```
Wrong type - expected address.
Wrong type - expected integer.
Wrong type for loop's END value - must match type of START value.
    These warnings are only given when type checking has been enabled
    using the "-Wtype-mismatch" switch. Make sure the argument type
    matches the instruction's addressing mode.
    In "!for" loops, START and END must have the same type, which then
    gets used for the loop counter.
```

```
Zeropage pointer wraps around from $ff to $00
    A zeropage-indirect addressing mode uses $ff as the argument. The
    6502 will then fetch the second pointer byte from $00 instead of
    $0100, therefore this warning is issued.
```

```
...called from here.
    If warnings and/or errors are output during a macro call, messages
    with this text are added to display the call stack (because you
    might need to fix the call instead of the macro itself).
```

Errors during assembly

```
"ACME" environment variable not found.
    This will be shown if the source code references any files from
    the library, but the library location variable wasn't set. This
    can only be given on systems using the said variable.
```

```
"!cbm" is obsolete; use "!ct pet" instead.
    This is given when the now obsolete "!cbm" pseudo opcode is
    encountered.
    If you want to assemble an old source code without first updating
    it, you can use the "--dialect" CLI switch to make ACME mimic an
    older version.
```

```
"!pseudopc/!realpc" is obsolete; use "!pseudopc {}" instead.
    This is given when one of the now obsolete !pseudopc/!realpc
    pseudo opcodes is encountered.
    If you want to assemble an old source code without first updating
    it, you can use the "--dialect" CLI switch to make ACME mimic an
    older version.
```

```
"!subzone {}" is obsolete; use "!zone {}" instead.
    This is given when the now obsolete "!subzone" pseudo opcode is
    encountered.
    If you want to assemble an old source code without first updating
    it, you can use the "--dialect" CLI switch to make ACME mimic an
    older version.
```

```
!error: ...
    This is given when the pseudo opcode "!error" is executed. The
    actual message varies according to the pseudo opcode's arguments.
```

```
After ELSE, expected block or IF/IFDEF/IFNDEF.
    There is something strange after ELSE: It must be "if", "ifdef",
    "ifndef" or an opening brace.
```

```
Argument out of range.
    You called arcsin/arccos with something not in the [-1, 1] range.
```

```
Cannot open input file.
    ACME had problems opening an input file ("!bin", "!convtab" or
    "!src"). Maybe you mistyped its name.
```

```
Conversion table incomplete.
    The conversion table file is too small. It needs to be exactly 256
    bytes in size.
```

```
Division by zero.
    Guess what - you attempted to divide by zero.
```

```
Expected ELSE or end-of-statement.
    There is something after the closing brace of an IF block that is
    not an ELSE.
```

```
Expected end-of-statement after ELSE block.
    There is something after the closing brace of an ELSE block.
```

```
Exponent is negative.
    Using negative exponents only give sensible results when using
    floating point maths.
```

```
Expression did not return a number.
    An expression returned a string or a list but a number (integer or
    float) was expected.
```

```
File name quotes not found ("" or <>).
    File names have to be given in quotes. Either "" quoting for files
    located in the current directory or <> quoting for library files.
```

```
Force bits can only be given to numbers.
    You tried to give a force bit to a symbol and then assign a string
    or list to it.
```

```
Found '}' instead of end-of-file.
    ACME encountered a '}' character when it expected the file to end
    instead (because no blocks were open).
```

```
Garbage data at end of statement (unexpected 'CHAR').
    There are still arguments when there should not be any more. The
    given character is the one where end-of-line was expected.
```

```
Hex digits are not given in pairs.
    The two digits of a hex byte are separated by another character,
    or there is an odd number of digits.
```

```
Illegal combination of command and addressing mode.
    The given command cannot be used with the given addressing mode on
    the CPU you have chosen.
```

```
Illegal combination of command and postfix.
    The given command cannot be used with the addressing mode
    indicated by the given postfix.
```

```
Illegal postfix.
    You used a postfix other than "+1", "+2" or "+3".
```

```
Index is undefined.
    You attempted an indexing operation with some undefined symbol.
```

```
Index out of range.
    The value for an indexing operation wasn't in the allowed range.
```

```
Macro already defined.
    Macros can only be defined once. If you define a macro twice, ACME
    will help you find the definitions by giving a warning for the
    first definition and a serious error (stopping assembly) for the
    second definition.
```

```
Macro not defined (or wrong signature).
    You tried to call a macro that either wasn't defined yet (always
```

define macros before using them) or was called with an illegal
argument list. There must be a 1:1 match between the definition's
formal parameters and the call's actual arguments.

Macro parameter twice.
The same symbol name is used two (or more) times in the same macro
parameter list.

Negative size argument.
The size argument of "!bin" or "!skip" must be zero or positive,
but cannot be negative.

Negative value - cannot choose addressing mode.
Because the argument is a negative value, ACME does not know what
addressing mode (8 bits, 16 bits, on a 65816 even 24 bits) to use.
You can overcome this problem using the postfix method. Or correct
your program to use positive addresses instead.

No string given.
ACME expects a string but doesn't find it, or the string is empty.

Number out of range.
A value is too high or too low.
This can also mean the desired addressing mode is not available,
as in "sty $e000, x".

```
Operation not supported: Cannot apply "OP" to "TYPE".
Operation not supported: Cannot apply "OP" to "TYPE" and "TYPE".
    You tried to use an operator on the wrong type(s) of argument(s),
    like indexing a float or negating a string.
```

```
Program counter is unset.
    You didn't set the program counter, so ACME didn't know where to
    start.
```

```
Quotes still open at end of line.
    You forgot the closing quotes.
```

```
Source file contains illegal character.
    Your source code file contained a null byte.
```

```
String length is not 1.
    You tried something like LDA#"X" with an illegal string length.
```

```
Symbol already defined.
    You defined a symbol that already had a different type or value.
    To change a symbol's type or value, use the "!set" pseudo opcode.
```

```
Syntax error.
    Guess what - there's a syntax error.
```

```
Target not in bank (0xTARGET).
    You tried to branch to an address not in the 0x0000..0xffff range.
    Relative addressing (branch commands or PER) cannot leave the
    current code bank of 64 KiB.
```

```
Target out of range (N; M too far).
    Branch commands use relative addressing, which only has a limited
    range. You exceeded it. N is the attempted offset, M is the
    difference to the limit - so if you succeed in optimizing M bytes
    away, the code would assemble.
```

```
The chosen CPU uses opcode 0xXY as a prefix code, do not use this mnemonic!
    The mnemonic is valid, but should not be used on this CPU. If you
    know better, you can get around this error like this:
    !cpu ANY_OTHER_CPU { PROBLEMATIC_MNEMONIC }
```

```
There's more than one character.
    You used a text string containing more than one character in a
    situation where only a string with length one is allowed.
```

```
Too late for postfix.
    You can only postfix symbols at the start, before they are used for
    the first time.
```

```
Too many '('.
    A formula ends before all parentheses were closed.
```

```
Un-pseudopc operator '&' can only be applied to labels.
    You tried to apply the operator '&' to something that is not a
    label. This operator only works on labels and on '*' (the program
    counter), it cannot be used on other objects.
```

```
Un-pseudopc operator '&' has no !pseudopc context.
    You either tried to apply the operator '&' to something that is
    not an implicitly defined label, but the result of an explicit
    symbol assignment (like the result of a calculation).
    Or you applied the operator to a label that was defined outside of
    a !pseudopc block, or, more generally, the number of '&'
    characters used was larger than the number of !pseudopc blocks
    around the definition.
```

```
Unknown encoding.
    You used the "!convtab" command with a keyword ACME does not know.
```

```
Unknown function.
    You used a mathematical function ACME does not know.
```

```
Unknown operator.
    You used an arithmetic/logical operator ACME does not know.
```

```
Unknown output format.
    You used the "!to" command with a keyword ACME does not know.
```

```
Unknown processor.
    You used the "!cpu" command with a keyword ACME does not know.
```

```
Unknown pseudo opcode.
    You have mistyped a "!" command.
```

```
Unknown "*=" segment modifier.
    You used a modifier keyword ACME does not know.
```

```
Unsupported backslash sequence.
    The character following the backslash was not one of the allowed
    ones. Backslash escaping was added in release 0.97 of ACME.
    If you want to assemble an old source code without first updating
    it, you can use the "--dialect" CLI switch to make ACME mimic an
    older version.
```

```
Unterminated index spec.
    An index was started with '[' but did not end with ']'.
```

```
Unterminated list.
    A list was started with '[' but did not end with ']'.
```

```
Value not defined (SYMBOL NAME).
    A value could not be worked out. Maybe you mistyped a symbol name.
```

```
    Whether this is given as a "normal" or as a serious error depends
    on the currently parsed pseudo opcode.
```

Serious errors (stopping assembly)

```
!serious: ...
    This is given when the pseudo opcode "!serious" is executed. The
    actual message varies according to the pseudo opcode's arguments.
```

```
Found end-of-file instead of '}'.
    The file ended when ACME expected the block to be closed instead
    (because there was at least one block left open).
```

```
Loop count is negative.
    You used the "!for" command with a negative loop count (getting
    this error is only possible when using the now deprecated syntax).
```

```
Macro already defined.
    Macros can only be defined once. If you define a macro twice, ACME
    will help you find both definitions by giving a warning for the
    first definition and a serious error (stopping assembly) for the
    second definition.
```

```
Missing '{'.
    ACME didn't find the expected '{' character. Remember that '{'
    characters must be given on the same line as the command they
    belong to.
```

```
Out of memory.
    When ACME runs out of memory, it stops assembly, giving this
    error. Free some memory and try again. It's highly unlikely anyone
    will ever see this error, though. ;)
```

```
Produced too much code.
    The program counter reached address $10000, leaving the output
    buffer. At the moment, ACME can only produce a maximum of 64 KB.
```

```
Syntax error.
    This is only given as a _serious_ error if it's in a "!do" loop
    condition.
```

```
Too deeply nested. Recursive macro calls?
    The only reason for ACME to have a limit on macro call nesting
    at all is to find infinite recursions.
    The default limit is 64, this can be changed using the
    "--maxdepth" CLI switch.
```

```
Too deeply nested. Recursive "!source"?
    The only reason for ACME to still have a limit on "!source"
    nesting at all is to find infinite recursions.
    The default limit is 64, this can be changed using the
    "--maxdepth" CLI switch.
```

```
Value not defined.
    A value could not be worked out. Maybe you mistyped a symbol name.
```

```
    Whether this is given as a "normal" or as a serious error depends
    on the currently parsed pseudo opcode.
```

Errors on closedown

```
Cannot open symbol list file "FILENAME".
Cannot open output file "FILENAME".
    Make sure the name doesn't contain wildcard characters and you
    have write access to the directory.
```

```
No output file specified (use the "-o" option or the "!to" pseudo opcode).
    You didn't specify the output file, so ACME did not create one.
```

Bugs in ACME

The warning "Bug in ACME, code follows" is always followed by a serious error message, stopping assembly. The second message actually gives a hint about the bug's location in the source code. If you ever get this combination of warning and serious error, please send me an e-mail and tell me about it. If possible, include a piece of source code that triggers it. Please don't get this wrong - there are no known bugs. I just left some debugging code in place in case there is a bug I failed to notice during testing. In practice, this warning is not expected to be given at all. That's the reason why I want to be notified if it *does* decide to show up.

The hint messages are of no real interest to the end user, but here they are for completeness' sake:

```
ArgStackEmpty
    There was no data for a monadic operator to work on.
```

```
ArgStackNotEmpty
    The expression parser has finished though there are still
    arguments left to process.
```

```
ExtendingListWithItself
    There were multiple references to the same list.
```

```
ForceBitZero
    The function to handle force bits was called without a force bit.
```

```
IllegalBlockTerminator
    A RAM block (macro or loop) was terminated incorrectly.
```

```
IllegalGroupIndex
    The mnemonic tree contains a group that I didn't add.
```

```
IllegalIfMode
    A sanity check in the if/ifdef/ifndef/else code failed.
```

```
IllegalImmediateMode
    The mnemonic tree contains invalid info about the size of immediate
    arguments.
```

```
IllegalInputSource
    Input is taken neither from a file nor from a RAM block.
```

IllegalNumberTypeX
    A number was neither INT nor FLOAT nor UNDEFINED.

IllegalObjectType
    A symbol is used that is neither number nor list nor string.

IllegalOperatorId
IllegalOperatorGroup
    The expression parser found an operator that does not exist.

IllegalSymbolNameLength
    A sanity check on string lengths failed.

NotEnoughArgs
    There was not enough data for a dyadic operator to work on.

NullTypeObject
ObjectHasNullType
    A symbol is used that does not have a type (number/list/string)
    associated with it.

OperatorIsNotDyadic
OperatorIsNotMonadic
    A function was passed the wrong type of operator.

```
OperatorStackNotEmpty
    The expression parser has finished though there are still
    operators left to process.
```

```
PartialEscapeSequence
    Buffered data ended on a backslash, which shouldn't be possible.
```

```
SecondArgIsNotAnInt
    A sanity check failed: An argument should have been converted to
    integer but wasn't.
```

```
StrangeInputMode
    The input state machine has reached a state that does not exist.
```

```
StrangeOperator
    The expression parser found a non-existing operator.
```

Undocumented Opcodes

Source: Illegals.txt.

In release 0.87, support for some of the undocumented opcodes of the NMOS 6502 processor was added. In release 0.89, some more were added. In release 0.94.8, another one was added (lxa). In release 0.95.3, C64DTV2 support was added, which includes these opcodes as well. In release 0.95.4, the remaining seven were added. In release 0.95.6, "ANC" was removed from C64DTV2 mode.

Here are the new mnemonics, possible addressing modes and generated opcodes (mnemonics in parentheses are used by other sources):

```
                |           addressing  mode           |
 mnemonic       |  8  8,x 8,y 16 16,x 16,y (8,x) (8),y | performs:
----------------+--------------------------------------+-----------
 slo (aso)      | 07  17      0f  1f   1b   03    13   | asl + ora
 rla (rln)      | 27  37      2f  3f   3b   23    33   | rol + and
 sre (lse)      | 47  57      4f  5f   5b   43    53   | lsr + eor
 rra (rrd)      | 67  77      6f  7f   7b   63    73   | ror + adc
 sax (axs, aax) | 87      97  8f           83         | stx + sta
 lax            | a7      b7  af       bf   a3    b3   | ldx + lda
 dcp (dcm)      | c7  d7      cf  df   db   c3    d3   | dec + cmp
 isc (isb, ins) | e7  f7      ef  ff   fb   e3    f3   | inc + sbc
 las (lar, lae) |                      bb             | A,X,S = {addr} & S
 tas (shs, xas) |                      9b             | S = A & X    {addr} = A&X& {H+1}
 sha (axa, ahx) |                      9f         93  | {addr} = A & X & {H+1}
 shx (xas, sxa) |                      9e             | {addr} = X & {H+1}
 shy (say, sya) |                 9c                  | {addr} = Y & {H+1}
```

```
                |         addressing mode         |
 mnemonic       | implied #8    8   8,x  16  16,x | performs:
----------------+---------------------------------+----------------------
 anc (ana, anb) |         0b*                     | A = A & arg, then C=N
 asr (alr)      |         4b                      | A = A & arg, then lsr
 arr            |         6b                      | A = A & arg, then ror
 sbx (axs, sax) |         cb                      | X = (A & X) - arg
 dop (nop, skb) |   80**  80   04   14            | skips next byte
 top (nop, skw) |   0c**                 0c   1c  | skips next two bytes
 nop (skb, skw) |   ea    80   04   14   0c   1c  | see the two lines above
 jam (kil, hlt) |   02                            | crash (wait for reset)
These two are somewhat unstable, because they involve an arbitrary value:
 ane (xaa, axm) |         8b***                   | A = (A | ??) & X & arg
 lxa (lax, atx) |         ab***                   | A,X = (A | ??) & arg
```

```
Example:
        !cpu nmos6502   ; activate additional mnemonics...
        lax (some_zp_label,x)   ; ...and use them. No, this
        dcp (other_zp_label),y  ; example does not make sense.
```

*) Up until ACME version 0.95.1, anc#8 generated opcode 0x2b. Since ACME version 0.95.2, anc#8 generates opcode 0x0b. Both opcodes work the same way on a real NMOS 6502 CPU, but they do not work on the C64DTV2. Using the "--dialect" CLI switch does not change the generated opcode!

**) Note that DOP ("double nop") and TOP ("triple nop") can be used with implied addressing, but the generated opcodes are those for immediate and 16-bit absolute addressing, respectively, This way DOP and TOP can be used to skip the following one- or two-byte instruction. Using DOP/TOP with x-indexed addressing might have its uses when timing is critical (crossing a page border adds a penalty cycle). Unless using implied addressing, DOP/TOP can also be written as NOP.

***) ANE and LXA first perform an ORA with an arbitrary(!) value and then perform an AND with the given argument. So they are unstable and therefore useless - unless the given argument is zero: ANE #0 reliably clears A - which is still useless; just use LDA #0. LXA #0 reliably clears both A and X. ACME will output a warning if these opcodes get assembled with a nonzero argument.

There is no guarantee that these opcodes actually work on a given 6502 (or 6510, or 8500, or 8501, or 8502) CPU. But as far as I know, nobody ever found an unmodified C64/C128 where these illegals didn't work.

These illegals will definitely *not* work on 65c02 and 65816 CPUs. But I really should not have to tell you that ;)

Because there are no official mnemonics for these opcodes, different people use different names for them. I hope my choices are not too exotic for your taste.

Just for the sake of completeness: Here are all the remaining opcodes (the ones ACME won't generate even with "nmos6502" cpu chosen):

```
Opcode|  Description                                          C64DTV2
------+-------------------------------------------------------------
  12  |  same as 02 and others    jam                CRASH   bra rel
  1a  |  same as (*legal*) ea     nop
  22  |  same as 02 and others    jam                CRASH
  2b  |  same as 0b               anc #8                     dop
  32  |  same as 02 and others    jam                CRASH   sac #8
  34  |  same as 14 and others    dop 8,x
  3a  |  same as (*legal*) ea     nop
  3c  |  same as 1c and others    top 16,x
  42  |  same as 02 and others    jam                CRASH   sir #8
  44  |  same as 04               dop 8
  52  |  same as 02 and others    jam                CRASH
  54  |  same as 14 and others    dop 8,x
  5a  |  same as (*legal*) ea     nop
  5c  |  same as 1c and others    top 16,x
  62  |  same as 02 and others    jam                CRASH
  64  |  same as 04               dop 8
  72  |  same as 02 and others    jam                CRASH
  74  |  same as 14 and others    dop 8,x
  7a  |  same as (*legal*) ea     nop
  7c  |  same as 1c and others    top 16,x
  82  |  same as c2/e2            dop #8, but said to CRASH sometimes
  89  |  same as 80               dop #8
  92  |  same as 02 and others    jam                CRASH
  b2  |  same as 02 and others    jam                CRASH
```

```
    c2  |  same as 82/e2           dop #8, but said to CRASH sometimes
    d2  |  same as 02 and others   jam                CRASH
    d4  |  same as 14 and others   dop 8,x
    da  |  same as (*legal*) ea    nop
    dc  |  same as 1c and others   top 16,x
    e2  |  same as 82/c2           dop #8, but said to CRASH sometimes
    eb  |  same as (*legal*) e9    sbc #8
    f2  |  same as 02 and others   jam                CRASH
    f4  |  same as 14 and others   dop 8,x
    fa  |  same as (*legal*) ea    nop
    fc  |  same as 1c and others   top 16,x
```

```
For more information about what these opcodes do, see these documents:
  John West, Marko MÃ¤kelÃ¤. '64doc' file, 1994/06/03.
  Extra Instructions Of The 65XX Series CPU, Adam Vardy, 27 Sept. 1996
  6502 Undocumented Opcodes, by Freddy Offenga, 5/17/1997
  AAY64 (All About Your 64)
```

...but the most comprehensive work is:

```
    "No More Secrets - NMOS 6510 Unintended Opcodes"
```

Download it from https://csdb.dk/release/?id=185341 or ask google for the latest version.

Floating-Point Expressions

Source: Floats.txt.

In release 0.92, ACME gained some experimental support for floating- point maths, so you can finally build sin/cos tables directly in ACME. But the expression parser still uses integer calculations per default. Here are the rules:

a) if a maths operation is useless when done with integers, it is done with floats and returns a float. Applies to sin(), cos(), tan(), arcsin(), arccos(), arctan() and float(): These are always computed in float mode and always return floats.

b) if a maths operation is useless when done with floats, it is done with integers and returns an integer. Applies to NOT, AND, OR, XOR, MOD, DIV, LSR, lowbyteof, highbyteof, bankbyteof and int(). These are always computed in integer mode and always return integers.

c) All other mathematical operations are done in float mode if and only if at least one of the operands is a float. So "1/2*.2*" *will give zero because it is done in integer mode, but* "*1.0/2*2" will give 1 because it is done in float mode.

To force a numerical value to be flagged as being a float, just add a decimal point and a zero. If a decimal value ends with a dot character, ACME switches to using the C type "double" and keeps reading digits. The value is then flagged internally as being float. CAUTION: Float usage is only activated when a decimal point has been found, so don't expect "10000000000000000000" to work. "10000000000000000000.0" won't work either, because when the decimal point gets parsed, the integer value will have overflown already. As there is no support yet for the "e" format (1e20 for 1*10^20) either, the only way to enter this value is by writing "1.0 * 10.0 ^ 20.0".

Examples:

```
!byte 1   / 2   * 2 ; gives 0 (integer maths)
!byte 1   / 2   * 2.0   ; gives 0 (1/2 => 0 in integer maths,
            ;   float usage comes too late)
!byte 1   / 2.0 * 2 ; gives 1 (FP in effect)
!byte 1   / 2.0 * 2.0   ; gives 1 (FP in effect)
!byte 1.0 / 2   * 2 ; gives 1 (FP in effect)
!byte 1.0 / 2   * 2.0   ; gives 1 (FP in effect)
!byte 1.0 / 2.0 * 2 ; gives 1 (FP in effect)
!byte 1.0 / 2.0 * 2.0   ; gives 1 (FP in effect)
```

You can use the new float() and int() functions to ensure the type of maths:

```
!byte a   / b   * c ; depends on a/b/c's internal flags
!byte float(a) / b * c  ; calculation is done as float
!byte int(a) / int(b) * int(c)  ; calculation is done as int
```

As you will have guessed, the trigonometric functions assume radians for measuring angles (90 degrees equals PI/2).

Have a look at the example source code "trigono.a", it builds some sin/cos tables.

65816-Specific Behavior

Source: 65816.txt.

This text contains information about the 65816-specific features of ACME.

Command aliases for "long" JMPs and JSRs

In addition to the commands JMP and JSR, the 65816 processor also knows JML and JSL, which are JMP and JSR using new (long) addressing modes. ACME also accepts the new addressing modes when using the old mnemonics JMP and JSR, but the old addressing modes cannot be used with the new mnemonics JML and JSL.

Argument order of MVN/MVP

```
According to WDC's official syntax for 65816 assembly language, the
argument order of the MVN and MVP instructions differs between
assembly language and machine code.
To copy bytes from bank $ab to bank $cd, use the following statement:
    mvn $ab, $cd        ; source bank $ab, destination bank $cd
or
    mvn #$ab, #$cd      ; source bank $ab, destination bank $cd
ACME will then produce the following machine code:
    $54 $cd $ab ; opcode mvn, destination bank $cd, source bank $ab
```

ACME 0.05 and earlier did it the wrong way.

Register lengths

When assembling "lda #5" for example, ACME has to know whether to create an 8-bit argument or a 16-bit argument. This depends on the current register length. On startup, ACME assumes all registers are 8 bits wide. You can change this at any time using the following pseudo opcodes:

```
        !al ; switch to long accumulator
        !as ; switch to short accumulator
        !rl ; switch to long index registers
        !rs ; switch to short index registers
```

Please note that ACME, unlike some other assemblers, does *not* track SEP/REP commands: I don't like that method - it fails when encountering PLPs, for example. So if it doesn't work reliably in the first place, why use it? :)

If you don't like that you always have to use a pseudo opcode alongside SEP/REP commands, then have a look at the file <65816/std.a> (in the library). There are some predefined macros that you can use.

Postfixing stuff

You can also use the postfix method (which is explained in the file "AddrModes.txt") to specify the immediate argument's length:

```
        ldx+2 #5
```

will always be assembled to a 16-bit argument, regardless of the currently configured index register width. Use at your own risk - this method obviously is not a good example on structured programming. :)

Miscellaneous

Note that ACME cannot produce more than 64 KBytes of code. Also note that though the 65816 CPU has an address space of 16 MB, ACME's program counter is only sixteen bits wide. It shouldn't be too hard to make any assembled code run in a non-zero bank, though.

General Description and Platforms

Source: Help.txt.

(C) 1998-2020 Marco Baye

Copyright

ACME - a crossassembler for producing 6502/65c02/65816 code. Copyright (C) 1998-2020 Marco Baye The ACME icon was designed by Wanja "Brix" Gayk

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Introduction

ACME is a crossassembler for the 65xx range of processors. It knows about the standard 6502, the 65c02 and the 65816. It also supports the undocumented ("illegal") opcodes of the NMOS versions of the 6502, like the 6510 variant that is used in the Commodore C=64, and it also supports extensions to the intruction set done by other parties.

This text and the other files in the same directory only describe the basic functions independent of the platform used. There should be another help file in this archive that outlines the features specific to your platform.

The files in the docs directory and what they contain:

```
65816.txt      Stuff specific to the 65816 processor
AddrModes.txt  How to choose non-standard addressing modes
AllPOs.txt     Lists ACME's pseudo opcodes. Use as a reference.
Changes.txt    The change log
COPYING        Version 2 of the GNU General Public License
Errors.txt     Lists ACME's error messages and what they mean.
Example.txt    Information on how to assemble the example sources.
Floats.txt     About the support for floating-point values
Help.txt       ...is this text.
Illegals.txt   Support for undocumented opcodes
Lib.txt        Information about the library
QuickRef.txt   All the basic stuff about ACME       <- START HERE!
```

```
    Source.txt     How to compile ACME
    Upgrade.txt    Incompatibilities to earlier versions
    cputypes/      Instruction sets of target CPUs
```

IMPORTANT: If you upgrade from an earlier version of ACME, don't forget to read the files "Changes.txt" and "Upgrade.txt". Adding new features can not always be done in a 100% compatible way, so newer version may behave slightly different. To solve this problem, the "--dialect" CLI switch can be used.

If you want to start using ACME right away, read the file "QuickRef.txt", it contains the main help text.

What it can and does

```
ACME is a crossassembler.
ACME can produce code for the 6502, 65c02 and 65816 processors.
It does this *fast*.
It can produce at most 64 KBytes of code.
You can use global labels, local labels and anonymous labels.
It is fast.
You can use global and local macros.
You can use conditional assembly.
You can use looping assembly.
You can include other source files.
You can include binary files (either whole or parts) directly into the
    output.
You can use offset assembly (code that is designed to run at a
    different address).
It is fast.
ACME's maths parser uses operator priorities, so 1+2*3 will correctly
    give 7 (unlike some other free assemblers that give 9 instead).
ACME's maths parser has no problems concerning parentheses and
    indirect addressing modes.
ACME's maths parser knows a shit load of different operations.
ACME supports both integer and floating point maths operations.
In addition to numbers, symbols can also hold strings or lists.
You can dump the global symbols into a file.
ACME supports a library of commonly used macros and symbols.
It always takes as many passes as are needed.
ACME exists on several platforms, meaning you can easily exchange your
    sources with other people (preferring other OSes).
ACME can convert its strings to PetSCII and screen code (Okay, this is
    C64-specific).
ACME has a rudimentary type checking system to catch errors like
    missing '#' characters.
Did I mention that it is fast?
```

What it can't and doesn't

ACME cannot transfer data to a C64 or another computer. ACME does not produce ".o65"-format linkable object files. ACME cannot produce more than 64 KB (would be useful for the 65816). ACME cannot disassemble or relocate given code files.

Platform independence

ACME was initially developed under RISC OS. Currently there are platform-specific versions available for AmigaOS, DOS, Linux, Windows and RISC OS. The Linux sources should be ready to compile on most other UNIX-like systems as well. In the future there will hopefully also be a version that runs on the C64/128. Though the source code does not exactly look like it *g*, ACME was written with portability in mind: Some of its limitations were included on purpose, just to allow a C64/128 version. To successfully assemble multi-file source codes from other platforms, the file names have to be altered as little as possible. Please name all your files that may be distributed in a sensible way, for example by limiting their file names to 8+3 format. I really hate this stupid will-it- ever-die DOS convention, but using it is the only way to ensure portability of files.

Please use ".a" as the file name extension of ACME source code files.

All file names used inside source code files have to be given in UNIX style, ACME will convert them to the current host platform style if needed.

There should be no problems concerning newline characters, ACME was designed to cope with CR, LF and CRLF.

A minor problem is the different character tables used on different systems. As all predefined ACME keywords only use 7-bit ASCII, the assembler will work on any system that uses a superset of this character table: UTF-8, ANSI, ISO 8859, etc. Symbol names can contain top-bit-set characters - these may look strange if the sources are edited on a different platform, but ACME will still work.

If you want to port ACME to another platform, please inform me so that I can add your version to the ones already present on the ACME homepage. As the sources are released under the GNU General Public License, you are not forced to do this; but it would help to make ACME available to other users. The same goes for any changes or enhancements to the sources: Please send me a copy so that the changes can be incorporated into the next "official" release on the ACME home page.

Contacting the author

The newest version of ACME can be found at the ACME homepage: http://sourceforge.net/p/acme-crossass/

If you want to report a bug or make a suggestion, then simply send me an email: mailto:marco@baye.de

🧩 Macro Library

> **Macro section**
> The entries in this section describe reusable ACME routines and helpers. Macro names, parameters, side effects, and required setup are highlighted so they can be scanned quickly.

Source: Lib.txt.

The files in the "ACME_Lib" directory tree can be accessed from within ACME sources when using the pseudo opcodes "!source" or "!binary":

If the file names are given using "..." quoting, ACME will look for the files in the current directory. If the file names are given using <...> quoting, ACME will look for them in the ACME_Lib directory tree however.

All files in the ACME_Lib directory tree are in the public domain. They are *NOT* covered by the GNU General Public License, under which the actual ACME program is released.

Building the Package Examples

Source: Example.txt.

To assemble the given example source code files, change to the "examples" directory and type

```
acme -DSYSTEM=64 ddrv.a
acme macedit.a
```

ACME will parse the source code files and will then produce files called "ddrv64.prg" and "macedit.o". You may compare them to the files called "ddrv64.exp" and "macedit.exp", to make sure ACME works as it should do.

Just in case you wonder:

```
"ddrv64.prg" is a joystick/mouse driver for the C64. The source
code is fairly well documented. Have a look at it if you need more
examples on how ACME works. By using "-DSYSTEM=128" instead of
"-DSYSTEM=64", you can also generate "ddrv128.prg", a C128 binary.
```

```
"macedit" is an unusably bad text editor for the C128. The source
code is not meant to be a good example of ACME's capabilities.
Please *don't* look at it. :)
```

```
"trigono.o" is a simple example written to test the floating-point
capabilities. Because floats are prone to rounding errors, there
are two different "expected" outputs: these were generated on
different architectures, they only differ in one byte ($7f/$80).
```

Building ACME from Source

Source: Source.txt.

This program is free software, released under the terms of the GNU General Public License. Therefore, the sources must be made publicly accessible. If this archive does not contain the source file tree, you can download it from the ACME web page at

http://sourceforge.net/p/acme-crossass/

To build ACME, you need a recent C compiler and a "make" utility. On most systems, typing "make" should suffice to build the binary. See your platform's help file for more information.

Compatibility and Version Changes

Source: Upgrade.txt.

If you haven't used ACME before, you don't need to read this text.

```
--------------------------------------------------------------------
Upgrading from earlier releases to ACME release 0.97
--------------------------------------------------------------------
```

```
a) Single quotes vs. double quotes:
Since "anything in double quotes" is now considered to be a string,
problems can arise when trying to do a calculation with a character
code. Here are some examples:
    lda #' '    ; loads 32 like before (ASCII code of space)
    lda #' ' + 1    ; loads 33 like before (32 plus one)
    lda #" "    ; loads 32 like before (ASCII code of space)
    lda #" " + 1    ; used to load 33, now fails with error!
The third example still works, because 1-char-strings are treated
just like single characters when returned by the expression parser as
an argument for a mnemonic.
However, the fourth example now fails because the expression parser
tries to add a string to an integer, which is an undefined operation.
Some examples for a related problem:
    a = ' '     ; a is 32 (ASCII code of space)
    b = ' ' + 1 ; b is 33 (32 plus one)
    c = "!"     ; c used to be 33, now it's a 1-char string
    d = "!" + "!"   ; d used to be 66, now it's a 2-char string
If you do not get any errors when compiling your old sources, you do
not need to worry about this problem.
If you _do_ get errors, just use single quotes instead of double
quotes. If you had to use double quotes because the quoted character
```

```
itself is a single quote, write '\'' instead (backslash escaping, see
below).
```

b) Backslash escaping: Backslashes in single or double quotes are now used as escape characters. You need to replace any backslash in older sources with a sequence of two backslashes, so "some\string" becomes "some\string", and a single character '\' becomes '\'. If you have used backslashes as directory separators in path names (in Windows/DOS environments), these also need changing - but instead of using a double backslash, just use a single forward slash ('/') instead. This has the added benefit of making the sources platform- independent (*and* it's compatible to older ACME releases as well).

```
c) Character values are now unsigned:
When parsing a character in single quotes, ACME returns its character
code, according to the chosen encoding (raw/petscii/screencode). If
this resulted in a byte with its most significant bit set, the actual
number was architecture-dependent. Here's an example:
        !ct pet ; choose PetSCII encoding
        x = 'A' ; PetSCII 'A' is 0xc1, so MSB is set
Now x was either -63 or +193, depending on the host cpu architecture.
Since release 0.97, this example will give 193 on all architectures.
In most cases, this is not a problem, because the actual bit pattern
of the lower eight bits is the same. But if you have written any code
where the numerical value of a PetSCII character is used for
computations _in_the_source_code_, please check those computations.
```

Use the "--dialect 0.94.12" CLI switch to get the old behavior concerning a) double quotes and b) backslashes. There is no way to get the old behavior concerning c) character values, because, as explained above, the old behavior was architecture-dependent, which is a bad idea(tm).

```
--------------------------------------------------------------------
Upgrading from earlier releases to ACME release 0.95.2
--------------------------------------------------------------------
```

In 6510 mode, ACME now outputs 0x0b instead of 0x2b when assembling the undocumented ("illegal") ANC #imm8 instruction. Both opcodes do the same thing, this was only changed because all other mnemonics use the lowest-numbered possible opcode as well. Forcing the old behavior via the "--dialect" switch is not supported.

```
--------------------------------------------------------------------
Upgrading from earlier releases to ACME release 0.94.12
--------------------------------------------------------------------
```

The pseudo opcode "!for" has a new syntax. The old syntax still works, but gives a warning. You can use the "--dialect 0.94.8" CLI switch to get the old behavior.

```
--------------------------------------------------------------------
Upgrading from earlier releases to ACME release 0.94.8
--------------------------------------------------------------------
```

The pseudo opcodes "!cbm", "!subzone" and "!realpc" no longer give warnings, but have now been disabled. You can use the "--dialect 0.94.6" CLI switch to get the old behavior.

```
--------------------------------------------------------------------
Upgrading from earlier releases to ACME release 0.94.6
--------------------------------------------------------------------
```

```
The "to-the-power-of" operator ('^') is now right-associative, so
    b^c^d    will now give    b^(c^d)    instead of    (b^c)^d
If you have never used the operator in this way, you don't need to
worry about it.
You can use the "--dialect 0.86" CLI switch to get the old behavior.
```

```
--------------------------------------------------------------------
Upgrading from earlier releases to ACME release 0.89
--------------------------------------------------------------------
```

```
The "logical shift right" operator has been changed. Note: This is
about ACME's expression parser and has nothing to do with the 6502
mnemonic called "LSR".
Older versions were supposed to work like this:
    a = b >> c  ; alias "LSR", do a logical shift right
But what they actually did depended on the compiler that was used to
create the ACME binary: many binaries did an "arithmetic shift right"
```

```
instead. This has now been fixed and changed to:
    a = b >> c  ; alias "ASR", do an arithmetic shift right
    a = b >>> c ; alias "LSR", do a logical shift right
If you have never applied the old ">>"/"LSR" operator to a negative
value, you do not need to worry about this. If you have, please check
what you expected to happen in those instances (arithmetic or logical
shift) and update your source codes accordingly (use either ">>"/"ASR"
or ">>>"/"LSR").
Forcing the old behavior via the "--dialect" switch is not possible,
because as explained above, the old behavior was compiler-dependent
anyway.
```

```
--------------------------------------------------------------------
Upgrading from earlier releases to ACME release 0.07
--------------------------------------------------------------------
```

Re-defining the program counter via "* = NEW_VALUE" no longer starts offset assembly. Instead, ACME will change its pointer into the output buffer to the given value, so you can write your code in distinct segments. These segments can be given in any order. After assembly, ACME stores everything from the lowest address used to the highest address used. Have a look at "AllPOs.txt" for an example on how to use this facility.

Offset assembly is now done using a new pseudo opcode called "!pseudopc". Have a look at "AllPOs.txt" for further information on its syntax and usage.

The pseudo opcode "!end" has been removed. Use "!eof" instead.

```
The mnemonic BIT can no longer be assembled without any argument. If
you want to insert the opcode only to mask the next instruction, use
    !src <6502/std.a>
to get the definitions for these two macros:
    +bit8   ; output $24 to mask following 1-byte command
    +bit16  ; output $2c to mask following 2-byte command
```

```
When using the 65816 cpu, ACME now uses the correct argument order for
the MVN and MVP mnemonics, which is:
    mnemonic source_bank, destination_bank
```

```
Using leading zeroes in hexadecimal or binary values makes ACME use
bigger addressing modes than needed. This has now been extended to
symbols as well:
    label2 = $00fa
        lda label2
will be assembled to:
    ad fa 00    lda $00fa
```

Forcing the old behavior via the "--dialect" switch is not supported.

```
--------------------------------------------------------------------------
Upgrading from earlier releases to ACME release 0.04 beta
--------------------------------------------------------------------------
```

The pseudo opcode "!module" has been removed. Use "!zone" instead. Forcing the old behavior via the "--dialect" switch is not supported.

```
--------------------------------------------------------------------------
Upgrading from earlier releases to ACME release 0.03 beta
--------------------------------------------------------------------------
```

It is no longer possible to have more than one label in a single line. Forcing the old behavior via the "--dialect" switch is not supported.

Change Log

Source: Changes.txt.

```
This text only contains descriptions of changes independent of the
platform used. There should be another help file in this archive
outlining the platform specific changes.
```

New in release 0.97

```
FINALLY strings can be assigned to symbols!
    "anything in double quotes" is a string, while characters in
    single quotes are, just as before, immediately converted to their
    character code. Go and read "docs/Upgrade.txt" to learn about
    compatibility issues.
Added backslash escaping in all string literals:
    \0 is a null byte, \t is a TAB, \n is a line feed, \r is a
    carriage return, \" is a double quote, \' is a single quote and
    \\ is a backslash. Go and read "docs/Upgrade.txt" to learn about
    compatibility issues.
Added "--dialect" CLI switch:
    Because string symbols and backslash escaping introduce a few
    incompatibilities to older versions, ACME can now be told to mimic
    the behavior of older versions. So it's still possible to assemble
    older sources.
Added lists:
    Lists can be used to pass an arbitrary number of arguments to
    macros, or to store any number of items (including other lists) in
    a single symbol. Example:   my_list = [1, 2, label, "string", 9]
Added "len()" operator:
    This returns the number of elements in a string or list.
Added "[]" operator (for indexing):
    This returns a single element from a string or list. Negative
    indices are supported as well, they access the string/list from
    the other end. Examples:   a = my_list[2]   b = my_string[-1]
Added "&" operator:
    This operator converts labels or the program counter from its
    value inside a "!pseudopc" block to the value outside of that
    block. Thanks to markusC64 for the suggestion!
Added "!while {}" pseudo opcode.
Added "else if", "else ifdef" and "else ifndef" possibilities.
Added "M65" cpu:
    This instruction set includes the MEGA65 extensions, namely 32-bit
    pointers and 32-bit data operations using prefix bytes.
Added "NMOS6502" as an alias for "6510" cpu.
Improved NMOS6502/6510 mode:
    DOP and TOP can now also be written as NOP.
Improved 65816 mode:
    MVN and MVP can now also be written with '#' before arguments.
Added "--test" CLI switch:
    This is for people who want to help test experimental features.
Improved error messages:
    "Garbage data at end of statement" now includes the unexpected
    character.
    "Symbol not defined" is only output once per symbol.
Added warning:
    ACME complains about binary literals with an "unusual" number of
    digits. Thanks to groepaz for the suggestion!
    The warning can be disabled using the "-Wno-bin-len" CLI switch.
Added warning:
    All mnemonics without indirect addressing now complain about
```

```
    unneeded parentheses.
Fix: Characters are now unsigned (had been architecture-dependent).
Improved error handling of "--cpu" and "--format" switches.
Added opcode table for NMOS6502 cpu to docs.
Added some test sources.
Added support for "hashbang" lines (if file starts with a '#'
    character, the first line is ignored)
Fixed some minor bugs no-one ever seems to have encountered.
Rewritten "docs/Upgrade.txt".
```

New in release 0.96.5

```


Allowed C++-style comments via "//". Thanks to awsm for the
    suggestion (and please accept my apologies for taking so long to
    release this)
Added "--ignore-zeroes" CLI switch. This disables the "leading zeroes
    determine number size" algorithm. Thanks to groepaz for the
    suggestion.
Added "--strict-segments" CLI switch. This changes warnings about
    overlapping memory segments into errors. Thanks to groepaz for the
    suggestion.
Added 6502 family tree to docs/cpu_types/all.txt
```

New in release 0.96.4

```


Bugfix: Removed warnings about zero page wrap-around for the 65816's
    24-bit pointers (because wrap-around does not actually happen).
    Thanks to Johann Klasek for reporting this.
Added "!xor" pseudo opcode to compensate for the shortcomings of the
    "!scrxor" pseudo opcode. Thanks to spider-j for the initial bug
    report.
Added "-I" CLI switch to add search paths for input files. Thanks to
    peiselulli for the suggestion.
```

New in release 0.96.3

```


Added "!h"/"!hex" pseudo opcode: Now external source code generator
    tools can easily put data in sources with minimal syntax overhead.
Added "!skip" pseudo opcode: "!skip N" works like "*=*+N" without
```

```
    starting a new segment.
Added "cheap locals": Labels with '@' prefix have automatic scoping,
    bounded by the preceding and the following global labels.
Added "--fullstop" CLI switch to change pseudo opcode prefix from '!'
    to '.' (so other assemblers' sources need less conversion work)
Fixed a bug where expressions like "1)+1" crashed ACME. Thanks to
    Bitbreaker for reporting this.
Added warning when using zp-indirect addressing modes where argument
    is $ff because pointer wraps around to $00. Thanks to Gerrit for
    the suggestion.
```

New in release 0.96.2

```
Added "--color" CLI switch to enable colored error output using ANSI
    escape codes. Thanks to Clifford Carnmo for submitting this patch.
```

New in release 0.96.1

```
Fixed bug where 65ce02's "(zp),z" addressing mode could be used in
    65816 mode.
```

New in release 0.96

```
Added experimental support for instruction sets of Rockwell 65C02,
    WDC 65C02(S), CSG 65CE02 and CSG 4502.
Stack indexing can now be given either as ",s" or as ",sp" (only
    relevant for 65816 and 65CE02).
```

New in release 0.95.8

```
Warnings and errors inside macros now cause the call stack to be
    displayed as well (does not yet work for serious errors, though).
```

New in release 0.95.7

```
New pseudo opcodes:
    "!be16", "!be24" and "!be32" for big-endian byte order output.
    "!le16", "!le24" and "!le32" for little-endian byte order output.
    The old pseudo opcodes ("!16", "!24", "!32") will now use the
    correct byte order for the chosen CPU (which is always little-
    endian, because there is no support for any big-endian CPU yet.;))
```

New in release 0.95.6

```
Fixed a bug: The "C64 DTV2" does not support the undocumented
    ("illegal") opcodes 0x0b and 0x2b, therefore the "ANC #8"
    mnemonic was removed from the "!cpu c64dtv2" mode. Thanks to
    peiselulli for testing and reporting this.
"Value not defined" error message now includes the name of the
    undefined symbol. Thanks to Thomas Woinke for suggesting this
    improvement.
Fixed a bug in type system: "!for" loop counters were not correctly
    flagged as "address" or "non-address".
The "addr()" function can now also be written as "address()".
Fixed a bug in report listing generator: CR characters in input caused
    additional blank lines in output. Thanks to Johann Klasek for
    submitting this patch.
```

New in release 0.95.5

```
Fixed a bug causing crashes on program exit (a pointer to a local
    struct kept being used later on, trashing the stack). Thanks to
    Hoeppie for reporting this.
```

New in release 0.95.4

```
Added the last remaining undocumented ("illegal") opcodes: LAS, TAS,
    SHA, SHX, SHY and ANE.
```

New in release 0.95.3

```
Added "c64dtv2" cpu type so you can use its SIR, SAC and BRA opcodes;
    along with the undocumented ("illegal") opcodes of the 6510.
Added "--msvc" CLI switch so error output can be configured to be in
    Visual Studio format. Thanks to Martin Piper for writing this
    patch!
Merged third-party patch (who wrote it?) to output label dump in VICE
    format. Still needs work to be configurable about the types of
    symbols actually output.
```

New in release 0.95.2

```
Changed "save labels" to "symbol list" in a lot of code, error
    messages and docs.
Added "!symbollist" alias for "!sl" pseudo opcode.
Change in undocumented ("illegal") opcodes: ANC #8 now generates 0x0b
    instead of 0x2b (both opcodes do the same thing).
Added experimental support for generating a report listing. Thanks to
    Johann Klasek for writing this extension patch.
```

New in release 0.95.1

```
Fixed a bug: Anonymous forward labels were allowed to be redefined.
Tweaked type system: When negating a value, address reference count
    will now be negated as well.
When using type system, label dump file will now contain "!addr" to
    mark addresses.
Changed "label" to "symbol" in a lot of code, error messages and docs.
```

New in release 0.95

```
Added an experimental type system to tell addresses and non-addresses
    apart (per default disabled; must be activated using the
    "-Wtype-mismatch" CLI switch). The new "!address" pseudo opcode is
    used to mark label definitions as setting address values.
Added "-Wno-old-for" switch to disable warning about old "!for"
    syntax. However, this will in turn enable warnings about using the
    *new* syntax.
```

New in release 0.94.12

```
Finally created new "!for" syntax so counting can start at zero. The
    old syntax still works, but gives a warning. See AllPOs.txt for
    more info.
Added "0b" as alternative prefix for binary values.
```

New in release 0.94.11

```
If !warn, !error and !serious are called with numbers, those will now
    be output in hex format as well. Also mentioned this in docs.
```

New in release 0.94.10

```
Warning about shift-space as first character of label name now works
    for UTF-8 as well.
Operator priority list in docs now contains built-in function calls.
```

New in release 0.94.9

```
C-style equality checking ("==" operator) is now recognized (but gives
    a warning). Thanks to groepaz for the suggestion.
```

New in release 0.94.8

```
Finally disabled pseudo opcode "!cbm". It now gives an error instead
    of a warning.
Finally disabled pseudo opcode "!realpc". It now gives an error
    instead of a warning. Therefore, "!pseudopc" now *must* be used
    with a block in {} braces.
Finally disabled pseudo opcode "!subzone". It now gives an error
    instead of a warning.
Added support for one more undocumented ("illegal") opcode: lxa. See
    Illegals.txt for more info.
Removed "Offset assembly still active at end of segment" warning. New
    segments can now be started even from within pseudopc sections.
```

New in release 0.94.7

```
Fixed bug: Setting the program counter to an address > $ffff and
    writing a byte hung ACME. Thanks to groepaz/VICEteam for reporting
    this.
Verbose output now uses "0x" instead of "$" to indicate hexadecimal in
    more places.
```

New in release 0.94.6

```
Made "power-of" operator right-associative.
```

New in release 0.94.5

```
Fixed bug: wrap-around branches (from $ff80+ to zp and back) were
    reported as errors. Thanks to Bitbreaker/VOZ for reporting this.
Fixed bug: left-shifting float values went wrong for large shifts.
New error message: "Target not in bank (0xTARGET)" for branches.
Fixed example program to get it to assemble with current library.
Streamlined "Floats.txt" docs.

Changes in library files:
<cbm/basicWHATEVER.a> files now contain opcode_* constants in addition
    to macros.
Added <cbm/mflpt.a>, containing a macro to store a float value in the
    five-byte format used by CBM basic (and while writing that macro,
    found the left-shift-float bug mentioned above, so the macro needs
    this fixed version to work).
Added <cbm/c64/float.a>, containing definitions for using the BASIC
    interpreter's float functions and constants.
```

New in release 0.94.4

```
Added "apple" output format.
```

New in release 0.94.3

```
"Target out of range" error now includes info on how much the range
    was exceeded. Thanks to Bitbreaker/VOZ for the suggestion.
```

New in release 0.94.2

```
Added missing newline after "no output file specified" message.
Fixed bug in "!to" and "!sl" (colon in filename is interpreted as
    command separator in later passes).
Changed verbose output hex prefix from $ to 0x.
Changed --help output
Changed EOR to XOR in docs and comments (the ACME operator, not the
    6502 opcode)
```

New in release 0.94.1

```
New CLI switch: "-D" allows to set global labels via the command line.
New CLI switch: "-Wno-label-indent" switches off warnings about
    indented implicit label definitions.
New PO: "!ifndef" (finally a companion for "!ifdef"...)
When setting the program counter via "* =", modifiers ("overlay" and
    "invisible") allow to suppress warnings about segment overlap.
Float values without leading digits are now accepted.
```

New in release 0.93

```
Change: If "Offset assembly still active at end of segment", it no
    longer gets switched off.
Change: Operators ASR and LSL/ASL now can also handle FP (LSR still
    makes no sense).
Change: Added distinction between '/' and "DIV" operators: DIV always
    gives integer results, while '/' depends on operands.
New functions: added int() and float() functions.
Internal change: default fill value for !align is now CPU-specific
    (but still 234)
New CLI switch: "--use-stdout" prints errors to stdout instead of
    stderr (a fix for the "Relaunch64" IDE I have nothing to do with)
```

New in release 0.92

```
Text versions of arithmetic/logic operators (XOR, DIV, MOD, etc.) no
    longer need to be in upper case.
Experimental support for floating point maths.
Support for mathematical functions:
    sin(), cos(), tan(), arcsin(), arccos(), arctan()
New errors:
    "Argument out of range.", "Unknown function."
These operators always deliver ints:
    not, and, or, xor, lowbyteof, highbyteof, bankbyteof, mod, asl,
    lsl, asr, lsr
```

New in release 0.91

```
Added anonymous labels (- + -- ++ --- +++ etc.). Every other assembler
    seems to support them, so I added them to ACME as well... :)
New POs: "!warn MESSAGE", "!error MESSAGE", "!serious MESSAGE"
New CLI option: "--maxdepth NUMBER" sets maximum recursion depth for
    macro calls and the "!source" pseudo opcode.
ACME now gives a warning when assembling JMP($xxff) on 6502/6510
    because that instruction is broken on those CPUs.
After giving the error "Target out of range", the error "Number out of
    range" is now suppressed.
Corrected code example in QuickRef.txt (why didn't anyone tell me? :))
Added additional example source code.
```

New in release 0.90

```
Arithmetic shift right now has some watchdog code and should work
    regardless of compiler.
Corrected some typos in error messages and docs.
New CLI option: "--cpu CPU_TYPE"
The output file format chosen with "--format FORMAT" is now used as
    default when "!to" is used without format keyword.
Again: Tidier code.
```

New in release 0.89

```
Support for more undocumented ("illegal") opcodes: anc, arr, asr, sbx,
    dop, top, jam. See Illegals.txt for more info.
Change in shift operators: Logical shift right (">>" or "LSR") has on
    most platforms actually been an arithmetic shift right all the
    time! Therefore, ">>" now *officially* performs an arithmetic
    shift right (can also be written as "ASR"), while ">>>" has been
    added to perform a logical shift right (can also be written as
    "LSR"). Note: This is about ACME's maths parser and has nothing to
    do with the 6502 mnemonics "asl" and "lsr".
Finally added a "-o" command line option to set the output file! See
    QuickRef.txt for info on the other new CLI options (--format,
    --labeldump, --maxerrors, --setpc, --initmem, --version).
Fixed bug: "!align" could be used while program counter undefined.
Fixed bug: Numbers before mnemonics are no longer skipped (or rather,
    implicit label definitions are no longer accepted if the label
```

```
    name starts with a digit).
Change: Much better algorithm to compute to-the-power-of (read: it's
    no longer braindead).
Some more internal tidying.
```

New in release 0.88

```
Fixed architecture-dependent bug introduced in release 0.87.
Fixed bug: Unknown !cpu keywords could cause crashes.
Fixed bug in !ct "filename" nesting.
```

New in release 0.87

```
Support for some undocumented ("illegal") opcodes: slo, rla, sre, rra,
    sax, lax, dcp, isc. To use these, choose the 6510 cpu.
Two error messages gone: "Sorry, feature not yet implemented." and
    "Chosen CPU does not support this command and/or addressing mode."
Explanation of new error message ("There's more than one character.")
    added to docs.
```

New in release 0.86

```
The "!convtab" pseudo opcode can now be given the file name of a
    conversion table. The file must hold exactly 256 bytes.
Improved docs a bit (more and better examples, more info on verbosity
    CLI switch).
If no "!to" pseudo opcode has been found, ACME will tell you so.
```

New in release 0.86 beta

```
Macros can now be used with call-by-reference semantics, therefore
    allowing some kind of return value. Call-by-reference is
    indicated by prefixing the relevant parameter(s) with a '~'
    character. This has to be done at both the macro definition and
```

```
    the macro call.
Different macros are allowed to have the same name as long as their
    parameter lists differ in size (number of arguments) or type
    (call-by-value vs. call-by-reference)
Macros do not have a limit on parameter count anymore.
Macro size is unlimited now.
The expression parser does not have a limit on recursion depth
    anymore, so you can use as many parentheses as you like.
Loop block size is unlimited now.
Label name and string lengths are unlimited now.
The recursion depth of "!source" and macro calls is set to 64. The
    only reason there still *is* a limit is to be able to spot
    infinite recursions.
Offset assembly now has block support and can be nested. Using the old
    syntax still works, but gives a warning.
Pseudo opcodes "!convtab", "!cpu", "!al", "!as", "!rl" and "!rs" now
    have block support and can be nested.
Using "!to" without file format indicator now gives a warning (but
    still works).
Fixed bug: The statement
    !to "outfile" ANY_SPECIAL_CHARACTER_BUT_COMMA GARBAGE
    wasn't flagged as an error.
Fixed bug: The statement
    !source "a file that cannot be opened"
    did not give an error, but was just ignored.
If a global label starts with a shift-space character, a warning is
    issued (because it is highly likely that it is a typing error).
*Much* cleaner internals. *Very* *much* cleaner internals actually.
More bug checking at runtime.
Tree lookups should be a bit faster.
Initialising the memory should be a bit faster.
Writing the output file should be a bit faster.
The expression parser now uses repeated multiplication instead of the
    math library's pow() call, so it is no longer necessary to include
    the C math library when compiling.
The number of errors displayed before assembly stops was reduced from
    20 to 10. I really should make this configurable via a CLI switch.
```

New in release 0.85 alpha

```
Fixed bug: Handling of parentheses in new expression parser was badly
    screwed up. Thanks go to Nathan Smith for reporting that bug.
Verbosity messages for segments and output file now contain size info.
```

New in release 0.84 alpha

```
Some changes in documentation (mainly corrected typos)
Usage count for labels (Unused ones are marked in label dump file)
New PO: "!8" (for 8-bit values, as "!byte" / "!by" / "!08")
Finally removed the dreaded only-two-input-files restriction
Improved PO: "!to" has parameter for choosing output file format
Fixed bug: Blanks after "!for"'s "}" character stopped assembly
Rewritten expression parser and label tree handler (should be faster)
Generally tidied up the source.
Skipped some version numbers to get a "less frightening" one. :)
```

New in release 0.08 beta

```
Fixed really serious bug: The 65816's indirect DP addressing caused
    wrong opcodes to be generated. Thanks to Doc Bacardi/The Dreams
    for reporting it.
```

New in release 0.07 beta

```
Fixed really serious bug: Indirect JMP / JSR were assembled without
    target addresses. Thanks to YTM/Alliance for reporting that one.
Fixed bug in value parser's handling of parentheses: Expressions like
    "a*(b-c)+d" gave "a*((b-c)+d)", obviously not the same.
Fixed bug: "!set LABEL = VALUE" now *really* works correctly.
Fixed bug: ACME gave "too late for postfix" error when reading a
    predefined label of known size. Only occurred when using macros.
Fixed bug: Error messages given from within macro definitions used
    truncated file names.
Fixed bug: Calling of local macros didn't work at all.
Fixed bug: "}" chars directly after macro calls were not found.
Fixed bug: Spaces after ":" and "{" gave syntax errors.
Fixed bug: Line counting inside loops was screwed up.
Fixed bug: Changed argument order of MVP and MVN (now it's "opcode,
    source, target")
New PO: "!08" (for 8-bit values, as "!byte" / "!by")
New PO: "!16" (for 16-bit values, as "!word" / "!wo")
New PO: "!24" (for 24-bit values)
New PO: "!32" (for *signed* 32-bit values)
New PO: "!pseudopc" (starts offset assembly)
New PO: "!realpc" (ends offset assembly)
New PO: "!for LABEL, TIMES { LINES }" for easier loops.
New PO: "!initmem BYTE" to define empty memory.
```

```
New PO: "!endoffile" (short "!eof") replaces "!end".
New PO: "!ifdef" (only use this if you *really* know what you are
    doing. Otherwise, just don't use it)
New PO: "!convtab CONVERSION" (short "!ct") selects the default
    character conversion, making "!cbm" obsolete.
Improved PO: "!binary" now has "skip" parameter.
Change: "!cbm" outputs a warning when used - use "!ct pet" instead.
Change: "!end" no longer works - use "!eof" instead.
Change: "* = VALUE" is now segment change instead of offset assembly.
Change: Argument order of MVN/MVP is now as is standard.
The typecast system has been rewritten - now it works as intended.
BIT without any parameters no longer works - use a macro instead.
Leading zeros are stored in label structures and acted upon.
The documentation is in several files now.
Negative numbers are now handled much more sensibly.
'ACME' environment variable only needed when *really* needed.
```

New in release 0.05 beta

```
Fixed bug: No more multiple error messages.
Fixed bug: Zone names now work correctly (First char wasn't stored).
Fixed bug: "!set label = label" now works correctly (I hope).
Fixed bug: "stz ...,y" gave "number too big" instead of "illegal
    combination of command and addressing mode"
New PO: "!subzone" (short "!sz") for nested zones.
Added support for library tree when using "!source" or "!binary".
Single-character strings can now be given in single quotes as well.
Real icons.
Startup errors now exit correctly with EXIT_FAILURE code.
Example program now includes "Expected_Output" file.
Further tidied up the sources.
Tidied up the general help file:
 -Changed "Freeware" to "free software"
 -Corrected the information given on "!align".
 -Added examples for most of the pseudo opcodes.
```

New in release 0.04 beta

```
Corrected some small bugs.
New PO: "!zone" (short "!zn") replaces "!module" (short "!mod")
Tidied up the sources a lot.
Changed bad style C code reported by lint.
Added GNU GPL hint in every source file.
Added startup message in verbose mode.
```

```
Added "Error: " to startup error messages.
Added Amiga, Linux and OS/2 versions
```

New in release 0.03 beta

```
Generally tidied up the source.
Moved RISC OS-specific CLI options to platform file.
Added pathname conversion from UNIX style to current platform style.
Added context variables (enabling "!source"s and macros).
Translated all documentation to english.
Changed string pseudo opcodes to allow numeric values.
Added verbose mode (CLI option "v").
Added output buffer, removing the need for additional output pass (and
    now the "!to" pseudo opcode can be placed anywhere).
More than one "label = pc" definition per statement now illegal.
Instead added possibility to have several statements on a single line
    by using ":" as a separator character.
Added new keywords: "!set", "!if", "else", "!do", "until", "while" and
    "!macro"
Added support for "!source".
Added basic support for blocks.
Added support for "!if {...} else {...}".
Added support for zone titles.
Added support for loops (endless loops are only detected if producing
    code).
Added support for macros (even nested definitions are possible now).
Added DOS version.
```

New in release 0.02 alpha

```
Er, I don't know anymore. It was a bad ugly hack and it only ran on
    RISC OS. :-)
```

CPU: all

Source: cputypes/all.txt.

```
ACME supports the following cpu types (shown here as a sort of family
tree):

 6502                   standard
 |
 |\_nmos6502 (=6510)    + undocumented opcodes
 |
 |\_c64dtv2             + BRA/SAC/SIR and some (not all!) undocumented
 |
  \_65c02               + BRA/PHX/PHY/PLX/PLY/STZ/TRB/TSB/...
    |
    |\_65816            16 bit regs, 24 bit address space, ...
    |
     \_r65c02           + bit manipulation instructions
       |
       |\_w65c02        + STP/WAI
       |
        \_65ce02        + Z reg, long branches, ...
          |
           \_4502       + MAP/EOM
             |
              \_m65     + 32-bit pointers, 32-bit 'Q' register



    !cpu 6502

This is the official instruction set of the original NMOS 6502 CPU
designed by MOS (later CSG).
There are 151 documented opcodes.
ACME does not use "A" to indicate "accumulator addressing"; just write
the mnemonic without any argument: "LSR" will work, "LSR A" won't.



    !cpu nmos6502

This instruction set includes the undocumented ("illegal") opcodes of
the NMOS 6502.
See "docs/Illegals.txt" for more info.



    !cpu 6510

This is an alias for "nmos6502", because the 6510 cpu (as used in the
C64 computer) is a variant of this type.



    !cpu c64dtv2
```

```
This is the cpu in version 2 of the C64DTV. It uses a superset of the
6502 instruction set. Features:
- new instructions:
    BRA near_target     branch always
    SAC #$12        set accumulator mapping
    SIR #$12        set index register mapping
- support for some (but not all!) of the undocumented opcodes.



    !cpu 65c02

This is the CMOS re-design of the 6502. It seems to have also been
available from Rockwell, GTE/CMD and others. Features:
- new instructions:
    BRA near_target     branch always
    PHX/PHY/PLX/PLY     push/pull X/Y register
    STZ $12         store zero in zp
    STZ $12, x      store zero in zp, x-indexed
    STZ $1234       store zero absolute
    STZ $1234, x        store zero absolute, x-indexed
    TRB $12         test and reset bits in zp
    TRB $1234       test and reset bits absolute
    TSB $12         test and set bits in zp
    TSB $1234       test and set bits absolute
- new addressing modes for existing instructions:
    LDA/STA/ADC/SBC ($12)   zp indirect
    AND/ORA/EOR/CMP ($12)   zp indirect
    BIT #$12        immediate
    BIT $12, x      zp, x-indexed
    BIT $1234, x        absolute, x-indexed
    INC         increment accumulator
    DEC         decrement accumulator
    JMP ($1234, x)      x-indexed indirect
- bugfix for flags in decimal mode
- bugfix for JMP($xxff) instruction
- undocumented opcodes are NOPs (although of different lengths)
There are 178 documented opcodes.



    !cpu 65816

This is a superset of 65c02, originally designed by WDC (it seems to
have been available from GTE/CMD as well). Features:
- register sizes can be changed to 16-bit
- 24-bit address space
- several new instructions (including block transfers)
- several new addressing modes for existing instructions
There are 256 documented opcodes, but one of them ("WDM") is reserved
for future expansion.
See "docs/65816.txt" for more info.
```

```
    !cpu r65c02
```

This is a superset of 65c02, probably originally by Rockwell. It adds bit manipulation instructions:

```
    BBR4 $12, near_target   branch on bit reset in zp
    BBS5 $12, near_target   branch on bit set in zp
    RMB6 $12        reset memory bit in zp
    SMB7 $12        set memory bit in zp
```

The digit in the mnemonic is the bit number, therefore it must be in the 0..7 range.
Chips with this instruction set seem to have been available from Rockwell, GTE/CMD and others.
There are 210 documented opcodes.

```
    !cpu w65c02
```

This is a superset of r65c02, originating at WDC. It adds two new instructions:

```
    STP         stop (wait for reset)
    WAI         wait for interrupt
```

There are 212 documented opcodes.

```
    !cpu 65ce02
```

This is a superset of r65c02, originating at CSG. Features:
- Z register
- 16-bit stack pointer
- 16-bit branches
- new instructions (including a few 16-bit operations)
- new addressing modes for existing instructions

There is a known bug: SBC does not work correctly in decimal mode.
There are 256 documented opcodes, but one of them ("AUG") is reserved for future expansion.
ACME uses different mnemonics for old and new (long) branch instructions:

```
    BEQ near_target     old, 8-bit offset
    LBEQ far_target     new, 16-bit offset
```

The original datasheet called BRA ("branch always") BRU ("branch unconditional") instead. ACME accepts both mnemonics.

```
    !cpu 4502
```

This is basically the same as 65ce02, but
- MAP replaces AUG
- EOM is synonymous to NOP

This cpu core can be found in the CSG4510 chip in the C65.

```
There are 256 documented opcodes.



    !cpu m65

This is a superset of 4502 specified by the MEGA65 project. It uses
NEG:NEG and NOP as prefix bytes to extend the instruction set.
Features:
- "quad mode" (32-bit data operations on virtual register 'Q')
- "long mode" (32-bit pointer addressing for existing mnemonics)
- "quad" and "long" modes can be combined
quad mode introduces several new mnemonics:
    LDQ/STQ/CPQ     like LDA/STA/CMP
    ADCQ/SBCQ       like ADC/SBC
    ANDQ/EORQ/ORQ       like AND/EOR/ORA
    ASLQ/LSRQ/ROLQ/RORQ like ASL/LSR/ROL/ROR
    INQ/DEQ         like INC/DEC
  The new mnemonics support all the addressing modes of the original
  mnemonics with two exceptions:
    - there are no 32-bit immediate arguments
    - indirect-Z-indexed addressing becomes indirect addressing
  CAUTION: The STQ instruction clobbers the N and Z flags!
  There is no "real" Q register, instead A/X/Y/Z are combined to form
  the Q register (A holds lsb, Z holds msb), except for read-modify-
  write instructions, where the 32-bit operation is performed without
  using A/X/Y/Z.
  To load a 32-bit immediate constant into the Q register, use the
  +movq macro from the <m65/std.a> library file.
long mode brings a single new addressing mode for eight mnemonics:
    LDA [$12], z        contents of $12/$13/$14/$15
    STA [$12], z        plus z form the address
    CMP [$12], z
    ADC [$12], z
    SBC [$12], z
    AND [$12], z
    EOR [$12], z
    ORA [$12], z
quad and long modes combined result in another addressing mode for
  eight of the new mnemonics:
    LDQ [$12]       contents of $12/$13/$14/$15
    STQ [$12]       form the address
    CPQ [$12]
    ADCQ [$12]
    SBCQ [$12]
    ANDQ [$12]
    EORQ [$12]
    ORQ [$12]
The NOP mnemonic is disabled for this instruction set because its
opcode is re-used internally as a prefix byte.
CAUTION: The !align pseudo opcode still inserts NOPs.
```

CPU: 4502

Source: cputypes/cpu 4502.txt.

```
  4502 opcode table

There are no more undocumented opcodes.
All differences to the 65ce02 are marked using '!' signs (only
opcodes 5c and ea anyway)


00  brk             01  ora (zp, x)     02  cle             03  see
04  tsb zp          05  ora zp          06  asl zp          07  rmb0 zp
08  php             09  ora #imm8       0a  asl             0b  tsy
0c  tsb abs16       0d  ora abs16       0e  asl abs16       0f  bbr0 zp, rel8
10  bpl rel8        11  ora (zp), y     12  ora (zp), z     13  bpl rel16
14  trb zp          15  ora zp, x       16  asl zp, x       17  rmb1 zp
18  clc             19  ora abs16, y    1a  inc             1b  inz
1c  trb abs16       1d  ora abs16, x    1e  asl abs16, x    1f  bbr1 zp, rel8

20  jsr abs16       21  and (zp, x)     22  jsr (abs16)     23  jsr (abs16, x)
24  bit zp          25  and zp          26  rol zp          27  rmb2 zp
28  plp             29  and #imm8       2a  rol             2b  tys
2c  bit abs16       2d  and abs16       2e  rol abs16       2f  bbr2 zp, rel8
30  bmi rel8        31  and (zp), y     32  and (zp), z     33  bmi rel16
34  bit zp, x       35  and zp, x       36  rol zp, x       37  rmb3 zp
38  sec             39  and abs16, y    3a  dec             3b  dez
3c  bit abs16, x    3d  and abs16, x    3e  rol abs16, x    3f  bbr3 zp, rel8

40  rti             41  eor (zp, x)     42  neg             43  asr
44  asr zp          45  eor zp          46  lsr zp          47  rmb4 zp
48  pha             49  eor #imm8       4a  lsr             4b  taz
4c  jmp abs16       4d  eor abs16       4e  lsr abs16       4f  bbr4 zp, rel8
50  bvc rel8        51  eor (zp), y     52  eor (zp), z     53  bvc rel16
54  asr zp, x       55  eor zp, x       56  lsr zp, x       57  rmb5 zp
58  cli             59  eor abs16, y    5a  phy             5b  tab
5c! map             5d  eor abs16, x    5e  lsr abs16, x    5f  bbr5 zp, rel8

60  rts             61  adc (zp, x)     62  rtn #imm8       63  bsr rel16
64  stz zp          65  adc zp          66  ror zp          67  rmb6 zp
68  pla             69  adc #imm8       6a  ror             6b  tza
6c  jmp (abs16)     6d  adc abs16       6e  ror abs16       6f  bbr6 zp, rel8
70  bvs rel8        71  adc (zp), y     72  adc (zp), z     73  bvs rel16
74  stz zp, x       75  adc zp, x       76  ror zp, x       77  rmb7 zp
78  sei             79  adc abs16, y    7a  ply             7b  tba
7c  jmp (abs16, x)  7d  adc abs16, x    7e  ror abs16, x    7f  bbr7 zp, rel8

80  bra rel8        81  sta (zp, x)     82  sta (o8, s), y  83  bra rel16
84  sty zp          85  sta zp          86  stx zp          87  smb0 zp
88  dey             89  bit #imm8       8a  txa             8b  sty abs16, x
8c  sty abs16       8d  sta abs16       8e  stx abs16       8f  bbs0 zp, rel8
```

```
90  bcc rel8        91  sta (zp), y     92  sta (zp), z     93  bcc rel16
94  sty zp, x       95  sta zp, x       96  stx zp, y       97  smb1 zp
98  tya             99  sta abs16, y    9a  txs             9b  stx abs16, y
9c  stz abs16       9d  sta abs16, x    9e  stz abs16, x    9f  bbs1 zp, rel8

a0  ldy #imm8       a1  lda (zp, x)     a2  ldx #imm8       a3  ldz #imm8
a4  ldy zp          a5  lda zp          a6  ldx zp          a7  smb2 zp
a8  tay             a9  lda #imm8       aa  tax             ab  ldz abs16
ac  ldy abs16       ad  lda abs16       ae  ldx abs16       af  bbs2 zp, rel8
b0  bcs rel8        b1  lda (zp), y     b2  lda (zp), z     b3  bcs rel16
b4  ldy zp, x       b5  lda zp, x       b6  ldx zp, y       b7  smb3 zp
b8  clv             b9  lda abs16, y    ba  tsx             bb  ldz abs16, x
bc  ldy abs16, x    bd  lda abs16, x    be  ldx abs16, y    bf  bbs3 zp, rel8

c0  cpy #imm8       c1  cmp (zp, x)     c2  cpz #imm8       c3  dew zp
c4  cpy zp          c5  cmp zp          c6  dec zp          c7  smb4 zp
c8  iny             c9  cmp #imm8       ca  dex             cb  asw abs16
cc  cpy abs16       cd  cmp abs16       ce  dec abs16       cf  bbs4 zp, rel8
d0  bne rel8        d1  cmp (zp), y     d2  cmp (zp), z     d3  bne rel16
d4  cpz zp          d5  cmp zp, x       d6  dec zp, x       d7  smb5 zp
d8  cld             d9  cmp abs16, y    da  phx             db  phz
dc  cpz abs16       dd  cmp abs16, x    de  dec abs16, x    df  bbs5 zp, rel8

e0  cpx #imm8       e1  sbc (zp, x)     e2  lda (o8, s), y  e3  inw zp
e4  cpx zp          e5  sbc zp          e6  inc zp          e7  smb6 zp
e8  inx             e9  sbc #imm8       ea! nop/eom         eb  row abs16
ec  cpx abs16       ed  sbc abs16       ee  inc abs16       ef  bbs6 zp, rel8
f0  beq rel8        f1  sbc (zp), y     f2  sbc (zp), z     f3  beq rel16
f4  phw #imm16      f5  sbc zp, x       f6  inc zp, x       f7  smb7 zp
f8  sed             f9  sbc abs16, y    fa  plx             fb  plz
fc  phw abs16       fd  sbc abs16, x    fe  inc abs16, x    ff  bbs7 zp, rel8


#imm8:  8-bit immediate value
#imm16: 16-bit immediate value (only for opcode f4)
zp:     8-bit zeropage address
abs16:  16-bit absolute address
rel8:   8-bit relative address offset
rel16:  16-bit relative address offset
o8:     8-bit offset (for stack-relative addressing, see opcodes 82 and e2)


The 4502 is a superset of the 65ce02. Opcode 5c (originally a "4-byte NOP
reserved for future expansion") has been changed to the "map" instruction,
now using implied addressing.
"map" uses the contents of the A/X/Y/Z registers to alter the memory map.
Because this might also change the current stack memory, all interrupts are
then implicitly disabled until an "eom" instruction gets executed.
"eom" stands for "end of mapping", but this is actually just a new alias for
opcode ea, the 6502-standard NOP.
```

CPU: 6502

Source: cputypes/cpu 6502.txt.

```
  6502 opcode table

All empty entries are undocumented opcodes.
Notice there are no documented opcodes in the fourth column.


00  brk             01  ora (zp, x)     02                  03
04                  05  ora zp          06  asl zp          07
08  php             09  ora #imm8       0a  asl             0b
0c                  0d  ora abs16       0e  asl abs16       0f
10  bpl rel8        11  ora (zp), y     12                  13
14                  15  ora zp, x       16  asl zp, x       17
18  clc             19  ora abs16, y    1a                  1b
1c                  1d  ora abs16, x    1e  asl abs16, x    1f

20  jsr abs16       21  and (zp, x)     22                  23
24  bit zp          25  and zp          26  rol zp          27
28  plp             29  and #imm8       2a  rol             2b
2c  bit abs16       2d  and abs16       2e  rol abs16       2f
30  bmi rel8        31  and (zp), y     32                  33
34                  35  and zp, x       36  rol zp, x       37
38  sec             39  and abs16, y    3a                  3b
3c                  3d  and abs16, x    3e  rol abs16, x    3f

40  rti             41  eor (zp, x)     42                  43
44                  45  eor zp          46  lsr zp          47
48  pha             49  eor #imm8       4a  lsr             4b
4c  jmp abs16       4d  eor abs16       4e  lsr abs16       4f
50  bvc rel8        51  eor (zp), y     52                  53
54                  55  eor zp, x       56  lsr zp, x       57
58  cli             59  eor abs16, y    5a                  5b
5c                  5d  eor abs16, x    5e  lsr abs16, x    5f

60  rts             61  adc (zp, x)     62                  63
64                  65  adc zp          66  ror zp          67
68  pla             69  adc #imm8       6a  ror             6b
6c  jmp (abs16)     6d  adc abs16       6e  ror abs16       6f
70  bvs rel8        71  adc (zp), y     72                  73
74                  75  adc zp, x       76  ror zp, x       77
78  sei             79  adc abs16, y    7a                  7b
7c                  7d  adc abs16, x    7e  ror abs16, x    7f

80                  81  sta (zp, x)     82                  83
84  sty zp          85  sta zp          86  stx zp          87
88  dey             89                  8a  txa             8b
8c  sty abs16       8d  sta abs16       8e  stx abs16       8f
90  bcc rel8        91  sta (zp), y     92                  93
```

```
94  sty zp, x       95  sta zp, x       96  stx zp, y       97
98  tya             99  sta abs16, y    9a  txs             9b
9c                  9d  sta abs16, x    9e                  9f

a0  ldy #imm8       a1  lda (zp, x)     a2  ldx #imm8       a3
a4  ldy zp          a5  lda zp          a6  ldx zp          a7
a8  tay             a9  lda #imm8       aa  tax             ab
ac  ldy abs16       ad  lda abs16       ae  ldx abs16       af
b0  bcs rel8        b1  lda (zp), y     b2                  b3
b4  ldy zp, x       b5  lda zp, x       b6  ldx zp, y       b7
b8  clv             b9  lda abs16, y    ba  tsx             bb
bc  ldy abs16, x    bd  lda abs16, x    be  ldx abs16, y    bf

c0  cpy #imm8       c1  cmp (zp, x)     c2                  c3
c4  cpy zp          c5  cmp zp          c6  dec zp          c7
c8  iny             c9  cmp #imm8       ca  dex             cb
cc  cpy abs16       cd  cmp abs16       ce  dec abs16       cf
d0  bne rel8        d1  cmp (zp), y     d2                  d3
d4                  d5  cmp zp, x       d6  dec zp, x       d7
d8  cld             d9  cmp abs16, y    da                  db
dc                  dd  cmp abs16, x    de  dec abs16, x    df

e0  cpx #imm8       e1  sbc (zp, x)     e2                  e3
e4  cpx zp          e5  sbc zp          e6  inc zp          e7
e8  inx             e9  sbc #imm8       ea  nop             eb
ec  cpx abs16       ed  sbc abs16       ee  inc abs16       ef
f0  beq rel8        f1  sbc (zp), y     f2                  f3
f4                  f5  sbc zp, x       f6  inc zp, x       f7
f8  sed             f9  sbc abs16, y    fa                  fb
fc                  fd  sbc abs16, x    fe  inc abs16, x    ff


#imm8:  8-bit immediate value
zp:     8-bit zeropage address
abs16:  16-bit absolute address
rel8:   8-bit relative address offset
```

CPU: 65816

Source: cputypes/cpu 65816.txt.

```
  65816 opcode table

There are no more undocumented opcodes.
All differences to the 65c02 are marked using a '+' sign, except
for the #imm8 -> #imm change.
```

```
00  brk             01  ora (dp, x)     02+ cop imm8        03+ ora offs8, s
04  tsb dp          05  ora dp          06  asl dp          07+ ora [dp]
08  php             09  ora #imm        0a  asl             0b+ phd
0c  tsb abs16       0d  ora abs16       0e  asl abs16       0f+ ora abs24
10  bpl rel8        11  ora (dp), y     12  ora (dp)        13+ ora (offs8, s), y
14  trb dp          15  ora dp, x       16  asl dp, x       17+ ora [dp], y
18  clc             19  ora abs16, y    1a  inc             1b+ tcs
1c  trb abs16       1d  ora abs16, x    1e  asl abs16, x    1f+ ora abs24, x

20  jsr abs16       21  and (dp, x)     22+ jsr abs24       23+ and offs8, s
24  bit dp          25  and dp          26  rol dp          27+ and [dp]
28  plp             29  and #imm        2a  rol             2b+ pld
2c  bit abs16       2d  and abs16       2e  rol abs16       2f+ and abs24
30  bmi rel8        31  and (dp), y     32  and (dp)        33+ and (offs8, s), y
34  bit dp, x       35  and dp, x       36  rol dp, x       37+ and [dp], y
38  sec             39  and abs16, y    3a  dec             3b+ tsc
3c  bit abs16, x    3d  and abs16, x    3e  rol abs16, x    3f+ and abs24, x

40  rti             41  eor (dp, x)     42+ wdm             43+ eor offs8, s
44+ mvp src, dst    45  eor dp          46  lsr dp          47+ eor [dp]
48  pha             49  eor #imm        4a  lsr             4b+ phk
4c  jmp abs16       4d  eor abs16       4e  lsr abs16       4f+ eor abs24
50  bvc rel8        51  eor (dp), y     52  eor (dp)        53+ eor (offs8, s), y
54+ mvn src, dst    55  eor dp, x       56  lsr dp, x       57+ eor [dp], y
58  cli             59  eor abs16, y    5a  phy             5b+ tcd
5c+ jmp abs24       5d  eor abs16, x    5e  lsr abs16, x    5f+ eor abs24, x

60  rts             61  adc (dp, x)     62+ per rel16       63+ adc offs8, s
64  stz dp          65  adc dp          66  ror dp          67+ adc [dp]
68  pla             69  adc #imm        6a  ror             6b+ rtl
6c  jmp (abs16)     6d  adc abs16       6e  ror abs16       6f+ adc abs24
70  bvs rel8        71  adc (dp), y     72  adc (dp)        73+ adc (offs8, s), y
74  stz dp, x       75  adc dp, x       76  ror dp, x       77+ adc [dp], y
78  sei             79  adc abs16, y    7a  ply             7b+ tdc
7c  jmp (abs16, x)  7d  adc abs16, x    7e  ror abs16, x    7f+ adc abs24, x

80  bra rel8        81  sta (dp, x)     82+ brl rel16       83+ sta offs8, s
84  sty dp          85  sta dp          86  stx dp          87+ sta [dp]
88  dey             89  bit #imm        8a  txa             8b+ phb
8c  sty abs16       8d  sta abs16       8e  stx abs16       8f+ sta abs24
90  bcc rel8        91  sta (dp), y     92  sta (dp)        93+ sta (offs8, s), y
94  sty dp, x       95  sta dp, x       96  stx dp, y       97+ sta [dp], y
98  tya             99  sta abs16, y    9a  txs             9b+ txy
9c  stz abs16       9d  sta abs16, x    9e  stz abs16, x    9f+ sta abs24, x

a0  ldy #imm        a1  lda (dp, x)     a2  ldx #imm        a3+ lda offs8, s
a4  ldy dp          a5  lda dp          a6  ldx dp          a7+ lda [dp]
a8  tay             a9  lda #imm        aa  tax             ab+ plb
ac  ldy abs16       ad  lda abs16       ae  ldx abs16       af+ lda abs24
b0  bcs rel8        b1  lda (dp), y     b2  lda (dp)        b3+ lda (offs8, s), y
b4  ldy dp, x       b5  lda dp, x       b6  ldx dp, y       b7+ lda [dp], y
b8  clv             b9  lda abs16, y    ba  tsx             bb+ tyx
bc  ldy abs16, x    bd  lda abs16, x    be  ldx abs16, y    bf+ lda abs24, x
```

```
c0  cpy #imm        c1  cmp (dp, x)     c2+ rep #imm8       c3+ cmp offs8, s
c4  cpy dp          c5  cmp dp          c6  dec dp          c7+ cmp [dp]
c8  iny             c9  cmp #imm        ca  dex             cb+ wai
cc  cpy abs16       cd  cmp abs16       ce  dec abs16       cf+ cmp abs24
d0  bne rel8        d1  cmp (dp), y     d2  cmp (dp)        d3+ cmp (offs8, s), y
d4+ pei (dp)        d5  cmp dp, x       d6  dec dp, x       d7+ cmp [dp], y
d8  cld             d9  cmp abs16, y    da  phx             db+ stp
dc+ jmp [abs16]     dd  cmp abs16, x    de  dec abs16, x    df+ cmp abs24, x

e0  cpx #imm        e1  sbc (dp, x)     e2+ sep #imm8       e3+ sbc offs8, s
e4  cpx dp          e5  sbc dp          e6  inc dp          e7+ sbc [dp]
e8  inx             e9  sbc #imm        ea  nop             eb+ xba
ec  cpx abs16       ed  sbc abs16       ee  inc abs16       ef+ sbc abs24
f0  beq rel8        f1  sbc (dp), y     f2  sbc (dp)        f3+ sbc (offs8, s), y
f4+ pea abs16       f5  sbc dp, x       f6  inc dp, x       f7+ sbc [dp], y
f8  sed             f9  sbc abs16, y    fa  plx             fb+ xce
fc+ jsr (abs16, x)  fd  sbc abs16, x    fe  inc abs16, x    ff+ sbc abs24, x


#imm:     immediate value (8 or 16 bits, depends on processor status)
#imm8:    8-bit immediate value
dp:       8-bit direct page address
offs8:    8-bit offset (for stack-relative addressing)
abs16:    16-bit absolute address
abs24:    24-bit absolute address
rel8:     8-bit relative address offset
rel16:    16-bit relative address offset
src, dst: two 8-bit bank values
    (CAUTION: assembler expects "mnemonic src, dst" syntax,
    but machine language order is actually "opcode dst src")


The instruction set of the 65816 is a much extended superset of that
of the 65c02. Among the improvements are:
- the register widths can be switched to 16 bits
- addresses can now be 24 bits wide
- zero page is now called direct page, with an arbitrary base address
- lots of new addressing modes
- block transfer instructions

New mnemonics
-------------
    02  cop imm8        coprocessor operation
    6b  rtl         return long (fetches 24-bit address from stack)
    82  brl rel16       branch long (16-bit offset)

    0b  phd         push direct page register
    4b  phk         push program bank register
    8b  phb         push data bank register
    62  per rel16       push effective relative address
    d4  pei (dp)        push effective indirect address
    f4  pea abs16       push effective absolute address

    2b  pld         pull direct page register
```

```
    ab  plb         pull data bank register

    1b  tcs         transfer C to stack pointer
    3b  tsc         transfer stack pointer to C
    5b  tcd         transfer C to direct page register
    7b  tdc         transfer direct page register to C
    ('C' means the whole 16-bit accumulator, even in 8-bit mode)
    9b  txy         transfer X to Y
    bb  tyx         transfer Y to X

    eb  xba         exchange high and low bytes of accumulator
    fb  xce         exchange Carry and Emulation bits

    c2  rep #imm8       clear bits in status register
    e2  sep #imm8       set bits in status register

    cb  wai         wait for interrupt
    db  stp         wait for reset

    42  wdm         (reserved for future expansion)

    block transfers:
    44  mvp src, dst    move previous (decrementing addresses)
    54  mvn src, dst    move next (incrementing addresses)
    the arguments are bank numbers. block size minus one must be in 16-bit
    accumulator. X holds source address, Y holds target address.
    after these instructions, data bank register is set to "dst".
    if blocks overlap:
    when moving a block to a higher address, use mvp and put the highest
    addresses in X/Y.
    when moving a block to a lower address, use mvn and put the lowest
    adresses in X/Y.

New addressing modes for existing mnemonics
-------------------------------------------
    22  jsr abs24
    5c  jmp abs24
    dc  jmp [abs16]
    fc  jsr (abs16, x)

    ora and eor adc sta lda cmp sbc
    -------------------------------
    03  23  43  63  83  a3  c3  e3  offs8, s
    07  27  47  67  87  a7  c7  e7  [dp]
    0f  2f  4f  6f  8f  af  cf  ef  abs24
    13  33  53  73  93  b3  d3  f3  (offs8, s), y
    17  37  57  77  97  b7  d7  f7  [dp], y
    1f  3f  5f  7f  9f  bf  df  ff  abs24, x
```

CPU: 65c02

Source: cputypes/cpu 65c02.txt.

```
  65c02 opcode table

All empty entries are undocumented opcodes.
Notice there are still no documented opcodes in the fourth column.
All differences to the original 6502 are marked using a '+' sign.


00  brk             01  ora (zp, x)     02                  03
04+ tsb zp          05  ora zp          06  asl zp          07
08  php             09  ora #imm8       0a  asl             0b
0c+ tsb abs16       0d  ora abs16       0e  asl abs16       0f
10  bpl rel8        11  ora (zp), y     12+ ora (zp)        13
14+ trb zp          15  ora zp, x       16  asl zp, x       17
18  clc             19  ora abs16, y    1a+ inc             1b
1c+ trb abs16       1d  ora abs16, x    1e  asl abs16, x    1f

20  jsr abs16       21  and (zp, x)     22                  23
24  bit zp          25  and zp          26  rol zp          27
28  plp             29  and #imm8       2a  rol             2b
2c  bit abs16       2d  and abs16       2e  rol abs16       2f
30  bmi rel8        31  and (zp), y     32+ and (zp)        33
34+ bit zp, x       35  and zp, x       36  rol zp, x       37
38  sec             39  and abs16, y    3a+ dec             3b
3c+ bit abs16, x    3d  and abs16, x    3e  rol abs16, x    3f

40  rti             41  eor (zp, x)     42                  43
44                  45  eor zp          46  lsr zp          47
48  pha             49  eor #imm8       4a  lsr             4b
4c  jmp abs16       4d  eor abs16       4e  lsr abs16       4f
50  bvc rel8        51  eor (zp), y     52+ eor (zp)        53
54                  55  eor zp, x       56  lsr zp, x       57
58  cli             59  eor abs16, y    5a+ phy             5b
5c                  5d  eor abs16, x    5e  lsr abs16, x    5f

60  rts             61  adc (zp, x)     62                  63
64+ stz zp          65  adc zp          66  ror zp          67
68  pla             69  adc #imm8       6a  ror             6b
6c  jmp (abs16)     6d  adc abs16       6e  ror abs16       6f
70  bvs rel8        71  adc (zp), y     72+ adc (zp)        73
74+ stz zp, x       75  adc zp, x       76  ror zp, x       77
78  sei             79  adc abs16, y    7a+ ply             7b
7c+ jmp (abs16, x)  7d  adc abs16, x    7e  ror abs16, x    7f

80+ bra rel8        81  sta (zp, x)     82                  83
84  sty zp          85  sta zp          86  stx zp          87
88  dey             89+ bit #imm8       8a  txa             8b
8c  sty abs16       8d  sta abs16       8e  stx abs16       8f
90  bcc rel8        91  sta (zp), y     92+ sta (zp)        93
94  sty zp, x       95  sta zp, x       96  stx zp, y       97
98  tya             99  sta abs16, y    9a  txs             9b
```

```
9c+ stz abs16       9d  sta abs16, x    9e+ stz abs16, x    9f

a0  ldy #imm8       a1  lda (zp, x)     a2  ldx #imm8       a3
a4  ldy zp          a5  lda zp          a6  ldx zp          a7
a8  tay             a9  lda #imm8       aa  tax             ab
ac  ldy abs16       ad  lda abs16       ae  ldx abs16       af
b0  bcs rel8        b1  lda (zp), y     b2+ lda (zp)        b3
b4  ldy zp, x       b5  lda zp, x       b6  ldx zp, y       b7
b8  clv             b9  lda abs16, y    ba  tsx             bb
bc  ldy abs16, x    bd  lda abs16, x    be  ldx abs16, y    bf

c0  cpy #imm8       c1  cmp (zp, x)     c2                  c3
c4  cpy zp          c5  cmp zp          c6  dec zp          c7
c8  iny             c9  cmp #imm8       ca  dex             cb
cc  cpy abs16       cd  cmp abs16       ce  dec abs16       cf
d0  bne rel8        d1  cmp (zp), y     d2+ cmp (zp)        d3
d4                  d5  cmp zp, x       d6  dec zp, x       d7
d8  cld             d9  cmp abs16, y    da+ phx             db
dc                  dd  cmp abs16, x    de  dec abs16, x    df

e0  cpx #imm8       e1  sbc (zp, x)     e2                  e3
e4  cpx zp          e5  sbc zp          e6  inc zp          e7
e8  inx             e9  sbc #imm8       ea  nop             eb
ec  cpx abs16       ed  sbc abs16       ee  inc abs16       ef
f0  beq rel8        f1  sbc (zp), y     f2+ sbc (zp)        f3
f4                  f5  sbc zp, x       f6  inc zp, x       f7
f8  sed             f9  sbc abs16, y    fa+ plx             fb
fc                  fd  sbc abs16, x    fe  inc abs16, x    ff


#imm8: 8-bit immediate value
zp:    8-bit zeropage address
abs16: 16-bit absolute address
rel8:  8-bit relative address offset


The 65c02 is the CMOS re-design of the 6502. It has a few
improvements:

New mnemonics
-------------
    test and (re-)set bits against accumulator:
        04  tsb zp
        0c  tsb abs16
        14  trb zp
        1c  trb abs16
    push/pull x/y:
        5a  phy
        7a  ply
        da  phx
        fa  plx
    store zero:
        64  stz zp
        74  stz zp, x
```

```
        9c  stz abs16
        9e  stz abs16, x
    branch always:
        80  bra rel8

New addressing modes for existing mnemonics
-------------------------------------------
    zp indirect addressing without indexing:
        12  ora (zp)
        32  and (zp)
        52  eor (zp)
        72  adc (zp)
        92  sta (zp)
        b2  lda (zp)
        d2  cmp (zp)
        f2  sbc (zp)
    implied ("accumulator") addressing:
        1a  inc
        3a  dec
    x-indexed indirect addressing:
        7c  jmp (abs16, x)
    three more addressing modes for BIT:
        34  bit zp, x
        3c  bit abs16, x
        89  bit #imm8
```

CPU: 65ce02

Source: cputypes/cpu 65ce02.txt.

```
  65ce02 opcode table

There are no more undocumented opcodes.
All differences to the r65c02 are marked using '+' or '!' signs:
'!' marks (backward-compatible) changes to existing instructions.
'+' marks new instructions.


00  brk             01  ora (zp, x)     02+ cle             03+ see
04  tsb zp          05  ora zp          06  asl zp          07  rmb0 zp
08  php             09  ora #imm8       0a  asl             0b+ tsy
0c  tsb abs16       0d  ora abs16       0e  asl abs16       0f  bbr0 zp, rel8
10  bpl rel8        11  ora (zp), y     12! ora (zp), z     13+ bpl rel16
14  trb zp          15  ora zp, x       16  asl zp, x       17  rmb1 zp
18  clc             19  ora abs16, y    1a  inc             1b+ inz
1c  trb abs16       1d  ora abs16, x    1e  asl abs16, x    1f  bbr1 zp, rel8

20  jsr abs16       21  and (zp, x)     22+ jsr (abs16)     23+ jsr (abs16, x)
```

```
24  bit zp          25  and zp          26  rol zp          27  rmb2 zp
28  plp             29  and #imm8       2a  rol             2b+ tys
2c  bit abs16       2d  and abs16       2e  rol abs16       2f  bbr2 zp, rel8
30  bmi rel8        31  and (zp), y     32! and (zp), z     33+ bmi rel16
34  bit zp, x       35  and zp, x       36  rol zp, x       37  rmb3 zp
38  sec             39  and abs16, y    3a  dec             3b+ dez
3c  bit abs16, x    3d  and abs16, x    3e  rol abs16, x    3f  bbr3 zp, rel8

40  rti             41  eor (zp, x)     42+ neg             43+ asr
44+ asr zp          45  eor zp          46  lsr zp          47  rmb4 zp
48  pha             49  eor #imm8       4a  lsr             4b+ taz
4c  jmp abs16       4d  eor abs16       4e  lsr abs16       4f  bbr4 zp, rel8
50  bvc rel8        51  eor (zp), y     52! eor (zp), z     53+ bvc rel16
54+ asr zp, x       55  eor zp, x       56  lsr zp, x       57  rmb5 zp
58  cli             59  eor abs16, y    5a  phy             5b+ tab
5c+ aug             5d  eor abs16, x    5e  lsr abs16, x    5f  bbr5 zp, rel8

60  rts             61  adc (zp, x)     62+ rtn #imm8       63+ bsr rel16
64! stz zp          65  adc zp          66  ror zp          67  rmb6 zp
68  pla             69  adc #imm8       6a  ror             6b+ tza
6c  jmp (abs16)     6d  adc abs16       6e  ror abs16       6f  bbr6 zp, rel8
70  bvs rel8        71  adc (zp), y     72! adc (zp), z     73+ bvs rel16
74! stz zp, x       75  adc zp, x       76  ror zp, x       77  rmb7 zp
78  sei             79  adc abs16, y    7a  ply             7b+ tba
7c  jmp (abs16, x)  7d  adc abs16, x    7e  ror abs16, x    7f  bbr7 zp, rel8

80  bra rel8        81  sta (zp, x)     82+ sta (o8, s), y  83+ bra rel16
84  sty zp          85  sta zp          86  stx zp          87  smb0 zp
88  dey             89  bit #imm8       8a  txa             8b+ sty abs16, x
8c  sty abs16       8d  sta abs16       8e  stx abs16       8f  bbs0 zp, rel8
90  bcc rel8        91  sta (zp), y     92! sta (zp), z     93+ bcc rel16
94  sty zp, x       95  sta zp, x       96  stx zp, y       97  smb1 zp
98  tya             99  sta abs16, y    9a  txs             9b+ stx abs16, y
9c! stz abs16       9d  sta abs16, x    9e! stz abs16, x    9f  bbs1 zp, rel8

a0  ldy #imm8       a1  lda (zp, x)     a2  ldx #imm8       a3+ ldz #imm8
a4  ldy zp          a5  lda zp          a6  ldx zp          a7  smb2 zp
a8  tay             a9  lda #imm8       aa  tax             ab+ ldz abs16
ac  ldy abs16       ad  lda abs16       ae  ldx abs16       af  bbs2 zp, rel8
b0  bcs rel8        b1  lda (zp), y     b2! lda (zp), z     b3+ bcs rel16
b4  ldy zp, x       b5  lda zp, x       b6  ldx zp, y       b7  smb3 zp
b8  clv             b9  lda abs16, y    ba  tsx             bb+ ldz abs16, x
bc  ldy abs16, x    bd  lda abs16, x    be  ldx abs16, y    bf  bbs3 zp, rel8

c0  cpy #imm8       c1  cmp (zp, x)     c2+ cpz #imm8       c3+ dew zp
c4  cpy zp          c5  cmp zp          c6  dec zp          c7  smb4 zp
c8  iny             c9  cmp #imm8       ca  dex             cb+ asw abs16
cc  cpy abs16       cd  cmp abs16       ce  dec abs16       cf  bbs4 zp, rel8
d0  bne rel8        d1  cmp (zp), y     d2! cmp (zp), z     d3+ bne rel16
d4+ cpz zp          d5  cmp zp, x       d6  dec zp, x       d7  smb5 zp
d8  cld             d9  cmp abs16, y    da  phx             db+ phz
dc+ cpz abs16       dd  cmp abs16, x    de  dec abs16, x    df  bbs5 zp, rel8

e0  cpx #imm8       e1  sbc (zp, x)     e2+ lda (o8, s), y  e3+ inw zp
```

```
e4  cpx zp          e5  sbc zp          e6  inc zp          e7  smb6 zp
e8  inx             e9  sbc #imm8       ea  nop             eb+ row abs16
ec  cpx abs16       ed  sbc abs16       ee  inc abs16       ef  bbs6 zp, rel8
f0  beq rel8        f1  sbc (zp), y     f2! sbc (zp), z     f3+ beq rel16
f4+ phw #imm16      f5  sbc zp, x       f6  inc zp, x       f7  smb7 zp
f8  sed             f9  sbc abs16, y    fa  plx             fb+ plz
fc+ phw abs16       fd  sbc abs16, x    fe  inc abs16, x    ff  bbs7 zp, rel8


#imm8:  8-bit immediate value
#imm16: 16-bit immediate value (only for opcode f4)
zp:     8-bit zeropage address
abs16:  16-bit absolute address
rel8:   8-bit relative address offset
rel16:  16-bit relative address offset
o8:     8-bit offset (for stack-relative addressing, see opcodes 82 and e2)


The 65ce02 is a superset of the r65c02. It has several improvements:

new Z register
--------------
The "stz" mnemonic no longer means "store zero", but "store Z register"
    (see opcodes 64, 74, 9c, 9e)
The "(zp)" addressing mode now becomes "(zp), z"
    (see opcodes 12, 32, 52, 72, 92, b2, d2, f2)
Z defaults to zero after reset, so these changes are backward compatible
until Z is loaded with a non-zero value.
New instructions for this register:
    1b  inz         increment Z
    3b  dez         decrement Z
    4b  taz         transfer A to Z
    6b  tza         transfer Z to A
    a3  ldz #imm8
    ab  ldz abs16
    bb  ldz abs16, x
    c2  cpz #imm8
    d4  cpz zp
    db  phz         push Z
    dc  cpz abs16
    fb  plz         pull Z

16-bit stack pointer
--------------------
    02  cle         clear stack extend disable
    03  see         set stack extend disable
    0b  tsy         transfer stack_ptr_high to Y
    2b  tys         transfer Y to stack_ptr_high

16-bit branches
---------------
    13  bpl rel16
    33  bmi rel16
    53  bvc rel16
```

```
    63  bsr rel16       relative jsr, "branch to subroutine"
    73  bvs rel16
    83  bra rel16       relative jmp
    93  bcc rel16
    b3  bcs rel16
    d3  bne rel16
    f3  beq rel16
To use these in ACME, use the mnemonics lbpl, lbmi, lbvc, ...
(except for "bsr", because there is no 8-bit version of it anyway)

new addressing modes for existing instructions
----------------------------------------------
    22  jsr (abs16)
    23  jsr (abs16, x)
    82  sta (offset8, s), y
    8b  sty abs16, x
    9b  stx abs16, y
    e2  lda (offset8, s), y

new instructions
----------------
    42  neg         negate A
    43  asr
    44  asr zp
    54  asr zp, x
    5b  tab
    5c  aug         "4-byte NOP reserved for future expansion"
    62  rtn #imm8
    7b  tba
    c3  dew zp
    cb  asw abs16
    e3  inw zp
    eb  row abs16
    f4  phw #imm16
    fc  phw abs16
```

CPU: m65

Source: cputypes/cpu m65.txt.

```
  m65 opcode table(s)

The m65 instruction set extends the 4502 instruction set using prefix bytes.
Therefore, the "normal" opcode table is the same as for the 4502 cpu (see that
file), so this file only contains information about the extensions.


"quad mode" allows 32-bit data operations using a virtual register called 'Q'.
```

The mnemonics aslq/lsrq/rolq/rorq/inq/deq have five addressing modes.
The mnemonics ldq/stq have nine addressing modes in quad mode, and a tenth when combined with long mode.
The mnemonics cpq/adcq/sbcq/andq/eorq/orq have eight addressing modes in quad mode, and a ninth when combined with long mode.
This mode is entered after a NEG:NEG (42 42) prefix, the following opcode is then taken from this table:

00	01 orq (zp, x)	02	03
04	05 orq zp	06 aslq zp	07
08	09	0a aslq	0b
0c	0d orq abs16	0e aslq abs16	0f
10	11 orq (zp), y	12 orq (zp)	13
14	15 orq zp, x	16 aslq zp, x	17
18	19 orq abs16, y	1a inq	1b
1c	1d orq abs16, x	1e aslq abs16, x	1f
20	21 andq (zp, x)	22	23
24	25 andq zp	26 rolq zp	27
28	29	2a rolq	2b
2c	2d andq abs16	2e rolq abs16	2f
30	31 andq (zp), y	32 andq (zp)	33
34	35 andq zp, x	36 rolq zp, x	37
38	39 andq abs16, y	3a deq	3b
3c	3d andq abs16, x	3e rolq abs16, x	3f
40	41 eorq (zp, x)	42	43
44	45 eorq zp	46 lsrq zp	47
48	49	4a lsrq	4b
4c	4d eorq abs16	4e lsrq abs16	4f
50	51 eorq (zp), y	52 eorq (zp)	53
54	55 eorq zp, x	56 lsrq zp, x	57
58	59 eorq abs16, y	5a	5b
5c	5d eorq abs16, x	5e lsrq abs16, x	5f
60	61 adcq (zp, x)	62	63
64	65 adcq zp	66 rorq zp	67
68	69	6a rorq	6b
6c	6d adcq abs16	6e rorq abs16	6f
70	71 adcq (zp), y	72 adcq (zp)	73
74	75 adcq zp, x	76 rorq zp, x	77
78	79 adcq abs16, y	7a	7b
7c	7d adcq abs16, x	7e rorq abs16, x	7f
80	81 stq (zp, x)	82 stq (zp, s), y	83
84	85 stq zp	86	87
88	89	8a	8b
8c	8d stq abs16	8e	8f
90	91 stq (zp), y	92 stq (zp)	93
94	95 stq zp, x	96	97
98	99 stq abs16, y	9a	9b
9c	9d stq abs16, x	9e	9f
a0	a1 ldq (zp, x)	a2	a3

```
a4                  a5 ldq zp           a6                  a7
a8                  a9                  aa                  ab
ac                  ad ldq abs16        ae                  af
b0                  b1 ldq (zp), y      b2 ldq (zp)         b3
b4                  b5 ldq zp, x        b6                  b7
b8                  b9 ldq abs16, y     ba                  bb
bc                  bd ldq abs16, x     be                  bf

c0                  c1 cpq (zp, x)      c2                  c3
c4                  c5 cpq zp           c6 deq zp           c7
c8                  c9                  ca                  cb
cc                  cd cpq abs16        ce deq abs16        cf
d0                  d1 cpq (zp), y      d2 cpq (zp)         d3
d4                  d5 cpq zp, x        d6 deq zp, x        d7
d8                  d9 cpq abs16, y     da                  db
dc                  dd cpq abs16, x     de deq abs16, x     df

e0                  e1 sbcq (zp, x)     e2 ldq (zp, s), y   e3
e4                  e5 sbcq zp          e6 inq zp           e7
e8                  e9                  ea                  eb
ec                  ed sbcq abs16       ee inq abs16        ef
f0                  f1 sbcq (zp), y     f2 sbcq (zp)        f3
f4                  f5 sbcq zp, x       f6 inq zp, x        f7
f8                  f9 sbcq abs16, y    fa                  fb
fc                  fd sbcq abs16, x    fe inq abs16, x     ff

zp:     8-bit zeropage address
abs16:  16-bit absolute address


"long mode" adds an addressing mode using 32-bit pointers for eight existing
mnemonics. This mode is entered after a NOP (ea) prefix, the following opcode
should then be one of these:

12 ora [zp], z      32 and [zp], z      52 eor [zp], z      72 adc [zp], z
92 sta [zp], z      b2 lda [zp], z      d2 cmp [zp], z      f2 sbc [zp], z


"quad" and "long" modes can be combined to have 32-bit data access using a
32-bit pointer. This adds another addressing mode for eight of the new
mnemonics. This mode is entered after a NEG:NEG:NOP (42 42 ea) prefix, the
following opcode should then be one of these:

12 orq [zp]         32 andq [zp]        52 eorq [zp]        72 adcq [zp]
92 stq [zp]         b2 ldq [zp]         d2 cpq [zp]         f2 sbcq [zp]


Because the addressing modes are changed a bit by the prefix codes, here are
some of the unsupported combinations just for comparison (these result in
"Illegal combination of command and addressing mode"):
    lda (zp)    ; 65c02 knew this, but 65ce02 added z index!
    lda [zp]    ; long mode also expects z index!
    ldq #imm    ; quad mode has no immediate addressing!
```

```
    ldq (zp), z ; quad mode does not use z index!
    ldq [zp], z ; quad and long modes combined do not use z index!
```

CPU: nmos6502

Source: cputypes/cpu nmos6502.txt.

```
  nmos6502 opcode table

This table includes all of the unintended ("illegal") opcodes. These are
marked using '+' or '!' signs:
'+' means the instruction is supported by ACME,
'!' means ACME will use a different (but functionally equivalent) opcode.


00  brk             01  ora (zp, x)     02+ jam             03+ slo (zp, x)
04+ nop zp          05  ora zp          06  asl zp          07+ slo zp
08  php             09  ora #imm8       0a  asl             0b+ anc #imm8
0c+ nop abs16       0d  ora abs16       0e  asl abs16       0f+ slo abs16
10  bpl rel8        11  ora (zp), y     12! jam             13+ slo (zp), y
14+ nop zp, x       15  ora zp, x       16  asl zp, x       17+ slo zp, x
18  clc             19  ora abs16, y    1a! nop             1b+ slo abs16, y
1c+ nop abs16, x    1d  ora abs16, x    1e  asl abs16, x    1f+ slo abs16, x

20  jsr abs16       21  and (zp, x)     22! jam             23+ rla (zp, x)
24  bit zp          25  and zp          26  rol zp          27+ rla zp
28  plp             29  and #imm8       2a  rol             2b! anc #imm8
2c  bit abs16       2d  and abs16       2e  rol abs16       2f+ rla abs16
30  bmi rel8        31  and (zp), y     32! jam             33+ rla (zp), y
34! nop zp, x       35  and zp, x       36  rol zp, x       37+ rla zp, x
38  sec             39  and abs16, y    3a! nop             3b+ rla abs16, y
3c! nop abs16, x    3d  and abs16, x    3e  rol abs16, x    3f+ rla abs16, x

40  rti             41  eor (zp, x)     42! jam             43+ sre (zp, x)
44! nop zp          45  eor zp          46  lsr zp          47+ sre zp
48  pha             49  eor #imm8       4a  lsr             4b+ asr #imm8
4c  jmp abs16       4d  eor abs16       4e  lsr abs16       4f+ sre abs16
50  bvc rel8        51  eor (zp), y     52! jam             53+ sre (zp), y
54! nop zp, x       55  eor zp, x       56  lsr zp, x       57+ sre zp, x
58  cli             59  eor abs16, y    5a! nop             5b+ sre abs16, y
5c! nop abs16, x    5d  eor abs16, x    5e  lsr abs16, x    5f+ sre abs16, x

60  rts             61  adc (zp, x)     62! jam             63+ rra (zp, x)
64! nop zp          65  adc zp          66  ror zp          67+ rra zp
68  pla             69  adc #imm8       6a  ror             6b+ arr #imm8
6c  jmp (abs16)     6d  adc abs16       6e  ror abs16       6f+ rra abs16
70  bvs rel8        71  adc (zp), y     72! jam             73+ rra (zp), y
74! nop zp, x       75  adc zp, x       76  ror zp, x       77+ rra zp, x
```

```
78  sei             79  adc abs16, y    7a! nop             7b+ rra abs16, y
7c! nop abs16, x    7d  adc abs16, x    7e  ror abs16, x    7f+ rra abs16, x

80+ nop #imm8       81  sta (zp, x)     82! nop #imm8       83+ sax (zp, x)
84  sty zp          85  sta zp          86  stx zp          87+ sax zp
88  dey             89! nop #imm8       8a  txa             8b+ ane #imm8
8c  sty abs16       8d  sta abs16       8e  stx abs16       8f+ sax abs16
90  bcc rel8        91  sta (zp), y     92! jam             93+ sha (zp), y
94  sty zp, x       95  sta zp, x       96  stx zp, y       97+ sax zp, y
98  tya             99  sta abs16, y    9a  txs             9b+ tas abs16, y
9c+ shy abs16, x    9d  sta abs16, x    9e+ shx abs16, y    9f+ sha abs16, y

a0  ldy #imm8       a1  lda (zp, x)     a2  ldx #imm8       a3+ lax (zp, x)
a4  ldy zp          a5  lda zp          a6  ldx zp          a7+ lax zp
a8  tay             a9  lda #imm8       aa  tax             ab+ lxa #imm8
ac  ldy abs16       ad  lda abs16       ae  ldx abs16       af+ lax abs16
b0  bcs rel8        b1  lda (zp), y     b2! jam             b3+ lax (zp), y
b4  ldy zp, x       b5  lda zp, x       b6  ldx zp, y       b7+ lax zp, y
b8  clv             b9  lda abs16, y    ba  tsx             bb+ las abs16, y
bc  ldy abs16, x    bd  lda abs16, x    be  ldx abs16, y    bf+ lax abs16, y

c0  cpy #imm8       c1  cmp (zp, x)     c2! nop #imm8       c3+ dcp (zp, x)
c4  cpy zp          c5  cmp zp          c6  dec zp          c7+ dcp zp
c8  iny             c9  cmp #imm8       ca  dex             cb+ sbx #imm8
cc  cpy abs16       cd  cmp abs16       ce  dec abs16       cf+ dcp abs16
d0  bne rel8        d1  cmp (zp), y     d2! jam             d3+ dcp (zp), y
d4! nop zp, x       d5  cmp zp, x       d6  dec zp, x       d7+ dcp zp, x
d8  cld             d9  cmp abs16, y    da! nop             db+ dcp abs16, y
dc! nop abs16, x    dd  cmp abs16, x    de  dec abs16, x    df+ dcp abs16, x

e0  cpx #imm8       e1  sbc (zp, x)     e2! nop #imm8       e3+ isc (zp, x)
e4  cpx zp          e5  sbc zp          e6  inc zp          e7+ isc zp
e8  inx             e9  sbc #imm8       ea  nop             eb! sbc #imm8
ec  cpx abs16       ed  sbc abs16       ee  inc abs16       ef+ isc abs16
f0  beq rel8        f1  sbc (zp), y     f2! jam             f3+ isc (zp), y
f4! nop zp, x       f5  sbc zp, x       f6  inc zp, x       f7+ isc zp, x
f8  sed             f9  sbc abs16, y    fa! nop             fb+ isc abs16, y
fc! nop abs16, x    fd  sbc abs16, x    fe  inc abs16, x    ff+ isc abs16, x


#imm8:  8-bit immediate value
zp:     8-bit zeropage address
abs16:  16-bit absolute address
rel8:   8-bit relative address offset
```

CPU: r65c02

Source: cputypes/cpu r65c02.txt.

```
  r65c02 opcode table

All empty entries are undocumented opcodes.
All differences to the 65c02 (all in the fourth column) are marked
using a '+' sign.


00  brk             01  ora (zp, x)     02                  03
04  tsb zp          05  ora zp          06  asl zp          07+ rmb0 zp
08  php             09  ora #imm8       0a  asl             0b
0c  tsb abs16       0d  ora abs16       0e  asl abs16       0f+ bbr0 zp, rel8
10  bpl rel8        11  ora (zp), y     12  ora (zp)        13
14  trb zp          15  ora zp, x       16  asl zp, x       17+ rmb1 zp
18  clc             19  ora abs16, y    1a  inc             1b
1c  trb abs16       1d  ora abs16, x    1e  asl abs16, x    1f+ bbr1 zp, rel8

20  jsr abs16       21  and (zp, x)     22                  23
24  bit zp          25  and zp          26  rol zp          27+ rmb2 zp
28  plp             29  and #imm8       2a  rol             2b
2c  bit abs16       2d  and abs16       2e  rol abs16       2f+ bbr2 zp, rel8
30  bmi rel8        31  and (zp), y     32  and (zp)        33
34  bit zp, x       35  and zp, x       36  rol zp, x       37+ rmb3 zp
38  sec             39  and abs16, y    3a  dec             3b
3c  bit abs16, x    3d  and abs16, x    3e  rol abs16, x    3f+ bbr3 zp, rel8

40  rti             41  eor (zp, x)     42                  43
44                  45  eor zp          46  lsr zp          47+ rmb4 zp
48  pha             49  eor #imm8       4a  lsr             4b
4c  jmp abs16       4d  eor abs16       4e  lsr abs16       4f+ bbr4 zp, rel8
50  bvc rel8        51  eor (zp), y     52  eor (zp)        53
54                  55  eor zp, x       56  lsr zp, x       57+ rmb5 zp
58  cli             59  eor abs16, y    5a  phy             5b
5c                  5d  eor abs16, x    5e  lsr abs16, x    5f+ bbr5 zp, rel8

60  rts             61  adc (zp, x)     62                  63
64  stz zp          65  adc zp          66  ror zp          67+ rmb6 zp
68  pla             69  adc #imm8       6a  ror             6b
6c  jmp (abs16)     6d  adc abs16       6e  ror abs16       6f+ bbr6 zp, rel8
70  bvs rel8        71  adc (zp), y     72  adc (zp)        73
74  stz zp, x       75  adc zp, x       76  ror zp, x       77+ rmb7 zp
78  sei             79  adc abs16, y    7a  ply             7b
7c  jmp (abs16, x)  7d  adc abs16, x    7e  ror abs16, x    7f+ bbr7 zp, rel8

80  bra rel8        81  sta (zp, x)     82                  83
84  sty zp          85  sta zp          86  stx zp          87+ smb0 zp
88  dey             89  bit #imm8       8a  txa             8b
8c  sty abs16       8d  sta abs16       8e  stx abs16       8f+ bbs0 zp, rel8
90  bcc rel8        91  sta (zp), y     92  sta (zp)        93
94  sty zp, x       95  sta zp, x       96  stx zp, y       97+ smb1 zp
98  tya             99  sta abs16, y    9a  txs             9b
9c  stz abs16       9d  sta abs16, x    9e  stz abs16, x    9f+ bbs1 zp, rel8
```

```
a0  ldy #imm8       a1  lda (zp, x)     a2  ldx #imm8       a3
a4  ldy zp          a5  lda zp          a6  ldx zp          a7+ smb2 zp
a8  tay             a9  lda #imm8       aa  tax             ab
ac  ldy abs16       ad  lda abs16       ae  ldx abs16       af+ bbs2 zp, rel8
b0  bcs rel8        b1  lda (zp), y     b2  lda (zp)        b3
b4  ldy zp, x       b5  lda zp, x       b6  ldx zp, y       b7+ smb3 zp
b8  clv             b9  lda abs16, y    ba  tsx             bb
bc  ldy abs16, x    bd  lda abs16, x    be  ldx abs16, y    bf+ bbs3 zp, rel8

c0  cpy #imm8       c1  cmp (zp, x)     c2                  c3
c4  cpy zp          c5  cmp zp          c6  dec zp          c7+ smb4 zp
c8  iny             c9  cmp #imm8       ca  dex             cb
cc  cpy abs16       cd  cmp abs16       ce  dec abs16       cf+ bbs4 zp, rel8
d0  bne rel8        d1  cmp (zp), y     d2  cmp (zp)        d3
d4                  d5  cmp zp, x       d6  dec zp, x       d7+ smb5 zp
d8  cld             d9  cmp abs16, y    da  phx             db
dc                  dd  cmp abs16, x    de  dec abs16, x    df+ bbs5 zp, rel8

e0  cpx #imm8       e1  sbc (zp, x)     e2                  e3
e4  cpx zp          e5  sbc zp          e6  inc zp          e7+ smb6 zp
e8  inx             e9  sbc #imm8       ea  nop             eb
ec  cpx abs16       ed  sbc abs16       ee  inc abs16       ef+ bbs6 zp, rel8
f0  beq rel8        f1  sbc (zp), y     f2  sbc (zp)        f3
f4                  f5  sbc zp, x       f6  inc zp, x       f7+ smb7 zp
f8  sed             f9  sbc abs16, y    fa  plx             fb
fc                  fd  sbc abs16, x    fe  inc abs16, x    ff+ bbs7 zp, rel8


#imm8: 8-bit immediate value
zp:    8-bit zeropage address
abs16: 16-bit absolute address
rel8:  8-bit relative address offset


The r65c02 is a superset of the 65c02. It adds bit manipulation instructions:
    smbB zp         set bit in zp location
    rmbB zp         reset bit in zp location
    bbsB zp, rel8       branch if bit is set in zp location
    bbrB zp, rel8       branch if bit is reset in zp location
The 'B' in the mnemonic is the bit number, therefore it must be in
the 0..7 range.
```

CPU: w65c02

Source: cputypes/cpu w65c02.txt.

```
  w65c02 opcode table

All empty entries are undocumented opcodes.
All differences to the r65c02 are marked using a '+' sign (only two
opcodes anyway: cb and db)


00  brk             01  ora (zp, x)     02                  03
04  tsb zp          05  ora zp          06  asl zp          07  rmb0 zp
08  php             09  ora #imm8       0a  asl             0b
0c  tsb abs16       0d  ora abs16       0e  asl abs16       0f  bbr0 zp, rel8
10  bpl rel8        11  ora (zp), y     12  ora (zp)        13
14  trb zp          15  ora zp, x       16  asl zp, x       17  rmb1 zp
18  clc             19  ora abs16, y    1a  inc             1b
1c  trb abs16       1d  ora abs16, x    1e  asl abs16, x    1f  bbr1 zp, rel8

20  jsr abs16       21  and (zp, x)     22                  23
24  bit zp          25  and zp          26  rol zp          27  rmb2 zp
28  plp             29  and #imm8       2a  rol             2b
2c  bit abs16       2d  and abs16       2e  rol abs16       2f  bbr2 zp, rel8
30  bmi rel8        31  and (zp), y     32  and (zp)        33
34  bit zp, x       35  and zp, x       36  rol zp, x       37  rmb3 zp
38  sec             39  and abs16, y    3a  dec             3b
3c  bit abs16, x    3d  and abs16, x    3e  rol abs16, x    3f  bbr3 zp, rel8

40  rti             41  eor (zp, x)     42                  43
44                  45  eor zp          46  lsr zp          47  rmb4 zp
48  pha             49  eor #imm8       4a  lsr             4b
4c  jmp abs16       4d  eor abs16       4e  lsr abs16       4f  bbr4 zp, rel8
50  bvc rel8        51  eor (zp), y     52  eor (zp)        53
54                  55  eor zp, x       56  lsr zp, x       57  rmb5 zp
58  cli             59  eor abs16, y    5a  phy             5b
5c                  5d  eor abs16, x    5e  lsr abs16, x    5f  bbr5 zp, rel8

60  rts             61  adc (zp, x)     62                  63
64  stz zp          65  adc zp          66  ror zp          67  rmb6 zp
68  pla             69  adc #imm8       6a  ror             6b
6c  jmp (abs16)     6d  adc abs16       6e  ror abs16       6f  bbr6 zp, rel8
70  bvs rel8        71  adc (zp), y     72  adc (zp)        73
74  stz zp, x       75  adc zp, x       76  ror zp, x       77  rmb7 zp
78  sei             79  adc abs16, y    7a  ply             7b
7c  jmp (abs16, x)  7d  adc abs16, x    7e  ror abs16, x    7f  bbr7 zp, rel8

80  bra rel8        81  sta (zp, x)     82                  83
84  sty zp          85  sta zp          86  stx zp          87  smb0 zp
88  dey             89  bit #imm8       8a  txa             8b
8c  sty abs16       8d  sta abs16       8e  stx abs16       8f  bbs0 zp, rel8
90  bcc rel8        91  sta (zp), y     92  sta (zp)        93
94  sty zp, x       95  sta zp, x       96  stx zp, y       97  smb1 zp
98  tya             99  sta abs16, y    9a  txs             9b
9c  stz abs16       9d  sta abs16, x    9e  stz abs16, x    9f  bbs1 zp, rel8
```

```
a0  ldy #imm8       a1  lda (zp, x)     a2  ldx #imm8       a3
a4  ldy zp          a5  lda zp          a6  ldx zp          a7  smb2 zp
a8  tay             a9  lda #imm8       aa  tax             ab
ac  ldy abs16       ad  lda abs16       ae  ldx abs16       af  bbs2 zp, rel8
b0  bcs rel8        b1  lda (zp), y     b2  lda (zp)        b3
b4  ldy zp, x       b5  lda zp, x       b6  ldx zp, y       b7  smb3 zp
b8  clv             b9  lda abs16, y    ba  tsx             bb
bc  ldy abs16, x    bd  lda abs16, x    be  ldx abs16, y    bf  bbs3 zp, rel8

c0  cpy #imm8       c1  cmp (zp, x)     c2                  c3
c4  cpy zp          c5  cmp zp          c6  dec zp          c7  smb4 zp
c8  iny             c9  cmp #imm8       ca  dex             cb+ wai
cc  cpy abs16       cd  cmp abs16       ce  dec abs16       cf  bbs4 zp, rel8
d0  bne rel8        d1  cmp (zp), y     d2  cmp (zp)        d3
d4                  d5  cmp zp, x       d6  dec zp, x       d7  smb5 zp
d8  cld             d9  cmp abs16, y    da  phx             db+ stp
dc                  dd  cmp abs16, x    de  dec abs16, x    df  bbs5 zp, rel8

e0  cpx #imm8       e1  sbc (zp, x)     e2                  e3
e4  cpx zp          e5  sbc zp          e6  inc zp          e7  smb6 zp
e8  inx             e9  sbc #imm8       ea  nop             eb
ec  cpx abs16       ed  sbc abs16       ee  inc abs16       ef  bbs6 zp, rel8
f0  beq rel8        f1  sbc (zp), y     f2  sbc (zp)        f3
f4                  f5  sbc zp, x       f6  inc zp, x       f7  smb7 zp
f8  sed             f9  sbc abs16, y    fa  plx             fb
fc                  fd  sbc abs16, x    fe  inc abs16, x    ff  bbs7 zp, rel8


#imm8: 8-bit immediate value
zp:    8-bit zeropage address
abs16: 16-bit absolute address
rel8:  8-bit relative address offset


The w65c02 is a superset of the r65c02. It only adds two instructions:
    cb wai      wait for interrupt
    db stp      wait for reset
```

Licenc – GNU GPL v2

Source: COPYING.

```
            GNU GENERAL PUBLIC LICENSE
               Version 2, June 1991

 Copyright (C) 1989, 1991 Free Software Foundation, Inc.
                       59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

  3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

    a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

    b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

    c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent

access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that

system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

  8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

  9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

  10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

                NO WARRANTY

  11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

  12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY

YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER
PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE
POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest
possible use to the public, the best way to achieve this is to make it
free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest
to attach them to the start of each source file to most effectively
convey the exclusion of warranty; and each file should have at least
the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) 19yy  <name of author>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this
when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License. Of course, the commands you use may
be called something other than `show w' and `show c'; they could even be
mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the program, if
necessary. Here is a sample; alter the names:

```
  Yoyodyne, Inc., hereby disclaims all copyright interest in the program
  `Gnomovision' (which makes passes at compilers) written by James Hacker.

  <signature of Ty Coon>, 1 April 1989
  Ty Coon, President of Vice

This General Public License does not permit incorporating your program into
proprietary programs.  If your program is a subroutine library, you may
consider it more useful to permit linking proprietary applications with the
library.  If this is what you want to do, use the GNU Library General
Public License instead of this License.
```

Source Files and Checksums

The original files are unchanged. SHA-256 values are calculated from the original file bytes.

Dokumentum	SHA-256
QuickRef.txt	c5124d41d265ce8642700f50c17dae5fb850588da5f5d5127bceeda49d68e44b
AllPOs.txt	f0e623f1a77e305d7dfc1d6a07d6086fa7ed06ee3316db69fd596d44b11427da
AddrModes.txt	a2fa139654f6525e430272bf26fe854b4389fee00513d606bec6341443665cde
Errors.txt	f63b36f7aa9f182556fc5311490fcb6aac85c6968c99e3104299627034c69f9f
Illegals.txt	81dbc1e8372445ce7b676ef4053dd66f30a51c80a1a667222f94d2435fe0dfef
Floats.txt	4c59e71234da92193946f4587dad577d043f8558aedb400053a97d0b1623198c
65816.txt	4d51421bfeceee198349515095971c3bfa5fc7aa366686ff18cd52d247450468
Help.txt	7313d5d3f80b2b0ce08d64854d9878eb556b30a8789dfb4ba89d0e3979799c42
Lib.txt	410cae84510f810bfa03e70301a8b494895c84abc9d1bb8a89f1f2cee3facf8c
Example.txt	29be947d0e26ca7d423b5850ace358543d3011e1bf743bb6e1037b451039e259
Source.txt	29a63f93fcc186c6bdb4b1a03fbf2d721ab6ba152ee3ecc4204eb9c4672825a9
Upgrade.txt	394f72ef13825f9aa916276eed767124fc01e594162af566a5f4fda8a1ac90ae
Changes.txt	74d499029f80d6d300d79b78b6cc076999c6365fa3fbce0fd5d4674aec122762
cputypes/all.txt	057ab60e978b068bd609e9bd5eb8fdadcf3aa4328b9a1caf084fd9e857dfb4f9
cputypes/cpu 4502.txt	3a886cf30a5d26ea88587c85bbf89341197f0a43e8816425e7b95b736ff1f88a
cputypes/cpu 6502.txt	5c96b68f99a078cd1abfe761f8dc381d282b79f393ec92d9543449f34ac32363
cputypes/cpu 65816.txt	fede00f640cf9991ce6d6384bf6d21131a7129753155d1a785054d96f7754955
cputypes/cpu 65c02.txt	0a3e64a3c01843fd7cc2c414474d162c98479d40f58b6585248ce6f52a514271
cputypes/cpu 65ce02.txt	72c3a3314447f0b508f236b82496b6519b785b4426155b64af6efab3509b24a1
cputypes/cpu m65.txt	7453f15f97ffc77100c38ffbb68ef39beef252021c53e716f70d8730dc3d4da4

Dokumentum	SHA-256
`cputypes/cpu nmos6502.txt`	`e7a661f8a251b50c63b94e299fc7c0367e01c5a90cb46fb23c4526af8830e82a`
`cputypes/cpu r65c02.txt`	`2754da7c68fb25d1204a33a670fc3c95a59cd2d6b610c22a1e9962992c0d2a0a`
`cputypes/cpu w65c02.txt`	`483681adf8e62d466f83436713168ec42742589a33dcf7a2a0fb57b08db56471`
`COPYING`	`dcc100d4161cc0b7177545ab6e47216f84857cda3843847c792a25289852dcaa`