64tass v1.51 r992 reference manual

This is the manual for 64tass, the multi pass optimizing macro assembler for the 65xx series of processors. Key features:

- Open source, mostly portable C with minimal dependencies
- Familiar syntax to Omicron TASS and TASM.
- Supports 6502, 65C02, R65C02, W65C02, 65CE02, 65816, DTV, 65EL02
- Arbitrary-precision integers and bitstrings, double precision floating point numbers
- Character and byte strings, array arithmetic
- Handles UTF-8, UTF-16 and 8 bit RAW encoded source files, unicode character strings
- Supports Unicode identifiers with case folding and compatibility normalization
- Built-in “linker” with section support
- CPU or flat address space for creating huge binaries (e.g. cartridges)
- Conditional compilation, macros, struct/union structures, scopes.

This is a development version, features or syntax may change over time. Not everything is backwards compatible.

Project page: http://sourceforge.net/projects/tass64/

1 Table of Contents

- **1 Table of Contents**
- **2 Usage tips**
- **3 Expressions and data types**
  - 3.1 Integer constants
  - 3.2 Bit string constants
  - 3.3 Floating point constants
  - 3.4 Character string constants
  - 3.5 Byte string constants
  - 3.6 Lists and tuples
  - 3.7 Dictionaries
  - 3.8 Code
  - 3.9 Addressing modes
  - 3.10 Uninitialized memory
  - 3.11 Booleans
  - 3.12 Types
  - 3.13 Symbols
    - 3.13.1 Regular symbols
    - 3.13.1 Local symbols
    - 3.13.1 Anonymous symbols
    - 3.13.1 Constant and re-definable symbols
    - 3.13.1 The star label
  - 3.14 Built-in functions
    - 3.14.1 Mathematical functions
    - 3.14.1 Other functions
  - 3.15 Expressions
    - 3.15.1 Operators
    - 3.15.1 Comparison operators
    - 3.15.1 Bit string extraction operators
    - 3.15.1 Conditional operators
    - 3.15.1 Address length forcing
    - 3.15.1 Compound assignment
- **4 Compiler directives**
  - 4.1 Controlling the compile offset and program counter

- 4.2 Dumping data
  - 4.2.1 Storing numeric values
  - 4.2.1 Storing string values
- 4.3 Text encoding
- 4.4 Structured data
  - 4.4.1 Structure
  - 4.4.1 Union
  - 4.4.1 Combined use of structures and unions
- 4.5 Macros
  - 4.5.1 Parameter references
  - 4.5.1 Text references
- 4.6 Custom functions
- 4.7 Conditional assembly
  - 4.7.1 If, else if, else
  - 4.7.1 Switch, case, default
- 4.8 Repetitions
- 4.9 Including files
- 4.10 Scopes
- 4.11 Sections
- 4.12 65816 related
- 4.13 Controlling errors
- 4.14 Target
- 4.15 Misc
- 4.16 Printer control

- **5 Pseudo instructions**
  - 5.1 Aliases
  - 5.2 Always taken branches
  - 5.3 Long branches
- **6 Original turbo assembler compatibility**
  - 6.1 How to convert source code for use with 64tass
  - 6.2 Differences to the original turbo ass macro on the C64
  - 6.3 Labels
  - 6.4 Expression evaluation
  - 6.5 Macros
  - 6.6 Bugs
- **7 Command line options**
  - 7.1 Output options
  - 7.2 Operation options
  - 7.3 Target selection on command line
  - 7.4 Source listing options
  - 7.5 Other options
- **8 Messages**
  - 8.1 Warnings
  - 8.2 Errors
  - 8.3 Fatal errors
- **9 Credits**
- **10 Default translation and escape sequences**
  - 10.1 Raw 8-bit source
    - 10.1.1 The none encoding for raw 8-bit
    - 10.1.1 The screen encoding for raw 8-bit
  - 10.2 Unicode and ASCII source
    - 10.2.1 The none encoding for Unicode
    - 10.2.1 The screen encoding for Unicode
- **11 Opcodes**
  - 11.1 Standard 6502 opcodes
  - 11.2 6502 illegal opcodes

- 11.3 65DTV02 opcodes
- 11.4 Standard 65C02 opcodes
- 11.5 R65C02 opcodes
- 11.6 W65C02 opcodes
- 11.7 W65816 opcodes
- 11.8 65EL02 opcodes
- 11.9 65CE02 opcodes
- **12 Appendix**
  - 12.1 Assembler directives
  - 12.2 Built-in functions
  - 12.3 Built-in types

2 Usage tips

64tass is a command line assembler, the source can be written in any text editor. As a minimum the source filename must be given on the command line. The "`-a`" parameter is highly recommended if the source is Unicode or ASCII.

```
64tass -a src.asm
```

There are also some useful parameters which are described later.

For comfortable compiling I use such "Makefile"s (for make):

```
demo.prg: source.asm macros.asm pic.drp music.bin
	64tass -C -a -B -i source.asm -o demo.tmp
	pucrunch -ffast -x 2048 demo.tmp >demo.prg
```

This way "demo.prg" is recreated by compiling "source.asm" whenever "source.asm", "macros.asm", "pic.drp" or "music.bin" had changed.

Of course it's not much harder to create something similar for win32 (make.bat), however this will always compile and compress:

```
64tass.exe -C -a -B -i source.asm -o demo.tmp
pucrunch.exe -ffast -x 2048 demo.tmp >demo.prg
```

Here's a slightly more advanced Makefile example with default action as testing in VICE, clean target for removal of temporary files and compressing using an intermediate temporary file:

```
all: demo.prg
	x64 -autostartprgmode 1 -autostart-warp +truedrive +cart $<

demo.prg: demo.tmp
	pucrunch -ffast -x 2048 $< >$@

demo.tmp: source.asm macros.asm pic.drp music.bin
	64tass -C -a -B -i $< -o $@

.INTERMEDIATE: demo.tmp
.PHONY: all clean
clean:
	$(RM) demo.prg demo.tmp
```

It's useful to add a basic header to your source files like the one below, so that the resulting file is directly runnable without additional compression:

```
*       = $0801
```

```
        .word (+), 2005  ;pointer, line number
        .null $9e, ^start;will be sys 4096
+       .word 0          ;basic line end

*       = $1000

start   rts
```

A frequently coming up question is, how to automatically allocate memory, without hacks like `*=*+1`? Sure there's `.byte` and friends for variables with initial values but what about zero page, or RAM outside of program area? The solution is to not use an initial value by using “`?`” or not giving a fill byte value to `.fill`.

```
*       = $02
p1      .word ?         ;a zero page pointer
temp    .fill 10        ;a 10 byte temporary area
```

Space allocated this way is not saved in the output as there's no data to save at those addresses.

What about some code running on zero page for speed? It needs to be relocated, and the length must be known to copy it there. Here's an example:

```
        ldx #size(zpcode)-1;calculate length
-       lda zpcode,x
        sta wrbyte,x
        dex             ;install to zeropage
        bpl -
        jsr wrbyte
        rts
;code continues here but is compiled to run from $02
zpcode  .logical $02
wrbyte  sta $ffff       ;quick byte writer at $02
        inc wrbyte+1
        bne +
        inc wrbyte+2
+       rts
        .here
```

The assembler supports lists and tuples, which does not seems interesting at first as it sound like something which is only useful when heavy scripting is involved. But as normal arithmetic operations also apply on all their elements at once, this could spare quite some typing and repetition.

Let's take a simple example of a low/high byte jump table of return addresses, this usually involves some unnecessary copy/pasting to create a pair of tables with constructs like `>(label-1)`.

```
jumpcmd lda hibytes,x   ; selected routine in X register
        pha
        lda lobytes,x   ; push address to stack
        pha
        rts             ; jump, rts will increase pc by one!
; Build an anonymous list of jump addresses minus 1
-       = (cmd_p, cmd_c, cmd_m, cmd_s, cmd_r, cmd_l, cmd_e)-1
lobytes .byte <(-)      ; low bytes of jump addresses
hibytes .byte >(-)      ; high bytes
```

There are some other tips below in the descriptions.

3 Expressions and data types

3.1 Integer constants

Integer constants can be entered as decimal digits of arbitrary length. An underscore can be used between digits as a separator for better readability of long numbers. The following operations are accepted:

x + y	add x to y	2 + 2 is 4
x - y	subtract y from x	4 - 1 is 3
x * y	multiply x with y	2 * 3 is 6
x / y	integer divide x by y	7 / 2 is 3
x % y	integer modulo of x divided by y	5 % 2 is 1
x ** y	x raised to power of y	2 ** 4 is 16
-x	negated value	-2 is -2
+x	unchanged	+2 is 2
~x	$-x-1$	~3 is -4
x | y	bitwise or	2 | 6 is 6
x ^ y	bitwise xor	2 ^ 6 is 4
x & y	bitwise and	2 & 6 is 2
x << y	logical shift left	1 << 3 is 8
x >> y	arithmetic shift right	-8 >> 3 is -1

Table 1: Integer operators and functions

Integers are automatically promoted to float as necessary in expressions. Other types can be converted to integer using the integer type `int`.

```
        .byte 23        ; decimal

        lda #((bitmap >> 10) & $0f) | ((screen >> 6) & $f0)
        sta $d018
```

3.2 Bit string constants

Bit string constants can be entered in hexadecimal form with a leading dollar sign or in binary with a leading percent sign. An underscore can be used between digits as a separator for better readability of long numbers. The following operations are accepted:

~x	invert bits	~%101 is ~%101
y .. x	concatenate bits	$a .. $b is $ab
y x n	repeat	%101 x 3 is %101101101
x[n]	extract bit(s)	$a[1] is %1
x[s]	slice bits	$1234[4:8] is $3
x | y	bitwise or	~$2 | $6 is ~$0
x ^ y	bitwise xor	~$2 ^ $6 is ~$4
x & y	bitwise and	~$2 & $6 is $4
x << y	bitwise shift left	$0f << 4 is $0f0
x >> y	bitwise shift right	~$f4 >> 4 is ~$f

Table 2: Bit string operators and functions

Length of bit string constants are defined in bits and is calculated from the number of bit digits used including leading zeros.

Bit strings are automatically promoted to integer or floating point as necessary in expressions. The higher bits are extended with zeros or ones as needed.

Other types can be converted to bit string using the bit string type `bits`.

```
        .byte $33       ; hex
        .byte %00011111 ; binary
        .text $1234     ; $34, $12

        lda $01
        and #~$07
        ora #$05
        sta $01

        lda $d015
        and #~%00100000 ;clear a bit
        sta $d015
```

3.3 Floating point constants

Floating point constants have a radix point in them and optionally an exponent. A decimal exponent is “e” while a binary one is “p”. An underscore can be used between digits as a separator for better readability. The following operations can be used:

`x + y`	add x to y	`2.2 + 2.2` is `4.4`
`x - y`	subtract y from x	`4.1 - 1.1` is `3.0`
`x * y`	multiply x with y	`1.5 * 3` is `4.5`
`x / y`	integer divide x by y	`7.0 / 2.0` is `3.5`
`x % y`	integer modulo of x divided by y	`5.0 % 2.0` is `1.0`
`x ** y`	x raised t power of y	`2.0 ** -1` is `0.5`
`-x`	negated value	`-2.0` is `-2.0`
`+x`	unchanged	`+2.0` is `2.0`
`x | y`	bitwise or	`2.5 | 6.5` is `6.5`
`x ^ y`	bitwise xor	`2.5 ^ 6.5` is `4.0`
`x & y`	bitwise and	`2.5 & 6.5` is `2.5`
`x << y`	logical shift left	`1.0 << 3.0` is `8.0`
`x >> y`	arithmetic shift right	`-8.0 >> 4` is `-0.5`
`~x`	almost −x	`~2.1` is almost `-2.1`

Table 3: Floating point operators and functions

As usual comparing floating point numbers for (non) equality is a bad idea due to rounding errors.

There are no predefined floating point constants, define them as necessary. Hint: pi is `rad(180)` and e is `exp(1)`.

Floating point numbers are automatically truncated to integer as necessary. Other types can be converted to floating point by using the type `float`.

Fixed point conversion can be done by using the shift operators. For example a 8.16 fixed point number can be calculated as `(3.14 << 16) & $ffffff`. The binary operators operate like if the floating point number would be a fixed point one. This is the reason for the strange definition of inversion.

```
        .byte 3.66e1       ; 36.6, truncated to 36
        .byte $1.8p4       ; 4:4 fixed point number (1.5)
        .int 12.2p8        ; 8:8 fixed point number (12.2)
```

3.4 Character string constants

Character strings are enclosed in single or double quotes and can hold any Unicode character. Operations like indexing or slicing are always done on the original representation. The current encoding is only applied when it's used in expressions as numeric constants or in context of text data directives. Doubling the quotes inside string literals escapes them and results in a single quote.

`y .. x`	concatenate strings	`"a" .. "b"` is `"ab"`
`y in x`	is substring of	`"b" in "abc"` is `true`
`a x n`	repeat	`"ab" x 3` is `"ababab"`
`a[i]`	character from start	`"abc"[1]` is `"b"`
`a[i]`	character from end	`"abc"[-1]` is `"c"`
`a[s]`	no change	`"abc"[:]` is `"abc"`
`a[s]`	cut off start	`"abc"[1:]` is `"bc"`
`a[s]`	cut off end	`"abc"[:-1]` is `"ab"`
`a[s]`	reverse	`"abc"[::-1]` is `"cba"`

Table 4: Character string operators and functions

Character strings are converted to integers, byte and bit strings as necessary using the current encoding and escape rules. For example when using a sane encoding `"z"-"a"` is `25`.

Other types can be converted to character strings by using the type `str` or by using the `repr` and `format` functions.

Indexing characters with positive integers start with zero. Negative indexes are translated internally by adding the number of characters to them, therefore −1 can be used to access the last character. Indexing with list of integers is possible as well so `"abc"[(-1, 0, 1)]` is `"cab"`.

Slicing is an operation when parts of string are extracted from a start position to an end position with a step value. These parameters are separated with colons enclosed in square brackets and are all optional. Their default values are [start:maximum:step=1]. Negative start and end characters are converted to positive internally by adding the length of string to them. Negative step operates in reverse direction, non single steps will jump over characters.

```
mystr   = "oeU"         ; text
        .text 'it''s'   ; text: it's
        .word "ab"+1    ; character, results in "bb" usually

        .text "text"[:2]     ; "te"
        .text "text"[2:]     ; "xt"
        .text "text"[:-1]    ; "tex"
        .text "reverse"[::-1]; "esrever"
```

3.5 Byte string constants

Byte strings are like character strings, but hold bytes instead of characters.

Quoted character strings prefixing by "b", "l", "n", "p" or "s" characters can be used to create byte strings. The resulting byte string contains what `.text`, `.shiftl`, `.null`, `.ptext` and `.shift` would create.

`y .. x`	concatenate strings	`b"a" .. b"b"` is `b"ab"`
`y in x`	is substring of	`b"b" in b"abc"` is `true`
`a x n`	repeat	`b"ab" x 3` is `b"ababab"`
`a[i]`	byte from start	`b"abc"[1]` is `b"b"`
`a[i]`	byte from end	`b"abc"[-1]` is `b"c"`
`a[s]`	no change	`b"abc"[:]` is `b"abc"`
`a[s]`	cut off start	`b"abc"[1:]` is `b"bc"`
`a[s]`	cut off end	`b"abc"[:-1]` is `b"ab"`
`a[s]`	reverse	`b"abc"[::-1]` is `b"cba"`

Table 5: Byte string operators and functions

Indexing and slicing works as with character strings.

Other types can be converted to byte strings by using the type `bytes`.

```
        .enc screen     ;use screen encoding
mystr   = b"oeU"        ;convert text to bytes, like .text
        .enc none       ;normal encoding

        .text mystr     ;text as originally encoded
        .text s"p1"     ;convert to bytes like .shift
        .text l"p2"     ;convert to bytes like .shiftl
        .text n"p3"     ;convert to bytes like .null
        .text p"p4"     ;convert to bytes like .ptext
```

3.6 Lists and tuples

Lists and tuples can hold a collection of values. Lists are defined from values separated by comma between square brackets `[1, 2, 3]`, an empty list is `[]`. Tuples are similar but are enclosed in parentheses instead. An empty tuple is `()`, a single element tuple is `(4,)` to differentiate from normal numeric expression parentheses. When nested they function similar to an array. Currently both types are immutable.

`y .. x`	concatenate lists	`[1] .. [2]` is `[1, 2]`
`y in x`	is member of list	`2 in [1, 2, 3]` is `true`
`a x n`	repeat	`[1, 2] x 2` is `[1, 2, 1, 2]`
`a[i]`	element from start	`("1", 2)[1]` is `2`
`a[i]`	element from end	`("1", 2, 3)[-1]` is `3`
`a[s]`	no change	`(1, 2, 3)[:]` is `(1, 2, 3)`
`a[s]`	cut off start	`(1, 2, 3)[1:]` is `(2, 3)`
`a[s]`	cut off end	`(1, 2.0, 3)[:-1]` is `(1, 2.0)`
`a[s]`	reverse	`(1, 2, 3)[::-1]` is `(3, 2, 1)`
`*a`	convert to arguments	`format("%d: %s", *mylist)`

Table 6: List and tuple operators and functions

Arithmetic operations are applied on the all elements recursively, therefore `[1, 2] + 1` is `[2, 3]`, and `abs([1, -1])` is `[1, 1]`.

Arithmetic operations between lists are applied one by one on their elements, so `[1, 2] + [3, 4]` is `[4, 6]`.

When lists form an array and columns/rows are missing the smaller array is stretched to fill in the gaps if possible, so `[[1], [2]] * [3, 4]` is `[[3, 4], [6, 8]]`.

Indexing elements with positive integers start with zero. Negative indexes are transformed to positive by adding the number of elements to them, therefor -1 is the last element. Indexing with list of integers is possible as well so `[1, 2, 3][(-1, 0, 1)]` is `[3, 1, 2]`.

Slicing is an operation when parts of list or tuple are extracted from a start position to an end position with a step value. These parameters are separated with colons enclosed in square brackets and are all optional. Their default values are [start:maximum:step=1]. Negative start and end elements are converted to positive internally by adding the number of elements to them. Negative step operates in reverse direction, non single steps will jump over elements.

```
mylist  = [1, 2, "whatever"]
mytuple = (cmd_e, cmd_g)

mylist  = ("e", cmd_e, "g", cmd_g, "i", cmd_i)
keys    .text mylist[::2]    ; keys ("e", "g", "i")
call_l  .byte <mylist[1::2]-1; routines (<cmd_e-1, <cmd_g-1, <cmd_i-1)
call_h  .byte >mylist[1::2]-1; routines (>cmd_e-1, >cmd_g-1, >cmd_i-1)
```

The `range(start, end, step)` built-in function can be used to create lists of integers in a range with a given step value. At least the end must be given, the start defaults to 0 and the step to 1. Sounds not very useful, so here are a few examples:

```
;Bitmask table, 8 bits from left to right
        .byte %10000000 >> range(8)
;Classic 256 byte single period sinus table with values of 0-255.
        .byte 128.5 + 127 * sin(range(256) * rad(360.0/256))
;Screen row address tables
-       = $400 + range(0, 1000, 40)
scrlo   .byte <(-)
scrhi   .byte >(-)
```

3.7 Dictionaries

Dictionaries are unsorted lists holding key and value pairs. Definition is done by collecting key:value pairs separated by comma between braces `{1:"value", "key":1, :"optional default value"}`.

Looking up a non existing key is normally an error unless a default value is given. An empty dictionary is `{}`. Currently this type is immutable. Numeric and string keys are accepted, the value can be anything.

`x[i]`	value lookup	`{"1":2}["1"]` is `2`
`y in x`	is a key	`1 in {1:2}` is `true`

Table 7: Dictionary operators and functions

```
        .text {1:"one", 2:"two"}[2]; "two"
```

3.8 Code

Code holds the result of compilation in binary and other enclosed objects. In an arithmetic operation it's used as the numeric address of the memory where it starts. The compiled content remains static even if later parts of the source overwrite the same memory area.

Indexing and slicing of code to access the compiled content might be implemented differently in future releases. Use this feature at your own risk for now, you might need to update your code later.

`a.b`	member	`label.locallabel`
`a[i]`	element from start	`label[1]`
`a[i]`	element from end	`label[-1]`
`a[s]`	copy as tuple	`label[:]`
`a[s]`	cut off start, as tuple	`label[1:]`
`a[s]`	cut off end, as tuple	`label[:-1]`
`a[s]`	reverse, as tuple	`label[::-1]`

Table 8: Label operators and functions

```
mydata  .word 1, 4, 3
mycode  .block
local   lda #0
        .bend

        ldx #size(mydata) ;6 bytes (3*2)
        ldx #len(mydata)  ;3 elements
        ldx #mycode[0]    ;lda instruction, $a9
        ldx #mydata[1]    ;2nd element, 4
        jmp mycode.local  ;address of local label
```

3.9 Addressing modes

Addressing modes are used for determining addressing modes of instructions.

For indexing there must be no white space between the comma and the register letter, otherwise the indexing operator is not recognized. On the other hand put a space between the comma and a single letter symbol in a list to avoid it being recognized as an operator.

`#`	immediate
`(`	indirect
`[`	long indirect
`,b`	data bank indexed
`,d`	direct page indexed
`,k`	program bank indexed
`,r`	data stack pointer indexed
`,s`	stack pointer indexed
`,x`	x register indexed
`,y`	y register indexed
`,z`	z register indexed

Table 9: Addressing mode operators

Parentheses are used for indirection and square brackets for long indirection. These operations are only available after instructions and functions to not interfere with their normal use in expressions.

Several addressing mode operators can be combined together. **Currently the complexity is limited to 3 operators. This is enough to describe all addressing modes of the supported CPUs.**

`#`	immediate	`lda #$12`
`#addr,#addr`	move	`mvp #5,#6`
`addr`	direct or relative	`lda $12 lda $1234 bne $1234`
`addr,addr`	direct page bit	`rmb 5,$12`
`addr,addr,addr`	direct page bit relative jump	`bbs 5,$12,$1234`
`(addr)`	indirect	`lda ($12) jmp ($1234)`
`(addr),y`	indirect y indexed	`lda ($12),y`
`(addr),z`	indirect z indexed	`lda ($12),z`
`(addr,x)`	x indexed indirect	`lda ($12,x) jmp ($1234,x)`
`[addr]`	long indirect	`lda [$12] jmp [$1234]`
`[addr],y`	long indirect y indexed	`lda [$12],y`
`addr,b`	data bank indexed	`lda 0,b`
`addr,b,x`	data bank x indexed	`lda 0,b,x`
`addr,b,y`	data bank y indexed	`lda 0,b,y`
`addr,d`	direct page indexed	`lda 0,d`
`addr,d,x`	direct page x indexed	`lda 0,d,x`
`addr,d,y`	direct page y indexed	`ldx 0,d,y`
`(addr,d)`	direct page indirect	`lda ($12,d)`
`(addr,d,x)`	direct page x indexed indirect	`lda ($12,d,x)`
`(addr,d),y`	direct page indirect y indexed	`lda ($12,d),y`
`(addr,d),z`	direct page indirect z indexed	`lda ($12,d),z`
`[addr,d]`	direct page long indirect	`lda [$12,d]`
`[addr,d],y`	direct page long indirect y indexed	`lda [$12,d],y`
`addr,k`	program bank indexed	`jsr 0,k`
`(addr,k,x)`	program bank x indexed indirect	`jmp ($1234,k,x)`
`addr,r`	data stack indexed	`lda 1,r`
`(addr,r),y`	data stack indexed indirect y indexed	`lda ($12,r),y`

Table 10: Valid addressing mode operator combinations

addr,s	stack indexed	lda 1,s
(addr,s),y	stack indexed indirect y indexed	lda ($12,s),y
addr,x	x indexed	lda $12,x
addr,y	y indexed	lda $12,y

Direct page, data bank, program bank indexed and long addressing modes of instructions are inteligently chosen based on the instruction type, the address ranges set up by .dpage, .databank and the current program counter address. Therefore the ",d", ",b" and ",k" indexing is only used in very special cases.

The direct page indexed addressing mode is not affected by the .dpage directive and always forces the 8 bit address as is. It's only usable for direct/zero page instructions.

The data bank indexed addressing mode is not affected by the .databank directive and always forces the 16 bit address as is. It's only usable with data bank accessing instructions.

The program bank indexed addressing mode is not affected by the current program bank and always generates the 16 bit constant value as is. It's only usable with jump instructions.

Normally addressing mode operators are used in expressions right after instructions. They can also be used for defining stack variable symbols when using a 65816, or to force a specific addressing mode.

```
param   = 1,s           ;define a stack variable
const   = #1            ;immediate constant
        lda 0,b         ;always "absolute" lda $0000
        lda param       ;results in lda $01,s
        lda param+1     ;results in lda $02,s
        lda (param),y   ;results in lda ($01,s),y
        ldx const       ;results in ldx #$01
```

3.10 Uninitialized memory

There's a special value for uninitialized memory, it's represented by a question mark. Whenever it's used to generate data it creates a "hole" where the previous content of memory is visible.

Uninitialized memory holes without previous content are not saved unless it's really necessary for the output format, in that case it's replaced with zeros.

It's not just data generation statements (e.g. .byte) that can create uninitialized memory, but .fill, .align, .offs or address manipulation as well.

```
*       = $200          ;bytes as necessary
        .word ?         ;2 bytes
        .fill 10        ;10 bytes
        .align 64       ;bytes as necessary
        .offs 16        ;16 bytes
```

3.11 Booleans

There are two predefined boolean variables, true and false.

In numeric expressions true is 1 and false is 0. Other types can be converted to boolean by using the type bool.

Booleans are created by comparison operators (<, <=, !=, ==, >=, >), logical operators (&&, ||, ^^, !), the membership operator (in) and the all and any functions.

Conditional expressions, logical expressions and conditional compilation uses them.

bits	At least one non-zero bit

Table 11: Boolean values of various types

bool	When true
bytes	At least one non-zero byte
code	Address is non-zero
float	Not 0.0
int	Not zero
str	At least one non-zero byte after translation

3.12 Types

The various types mentioned earlier have predefined names. These can used for conversions or type checks.

address	Address type
bits	Bit string type
bool	Boolean type
bytes	Byte string type
code	Code type
dict	Dictionary type
float	Floating point type
gap	Uninitialized memory type
int	Integer type
list	List type
str	Character string type
tuple	Tuple type
type	Type type

Table 12: Built-in type names

```
        .cerror type(var) != str, "Not a string!"
        .text str(year)    ; convert to string
```

3.13 Symbols

Symbols are used to reference objects. Regularly named, anonymous and local symbols are supported. These can be constant or re-definable.

Scopes are where symbols are stored and looked up. The global scope is always defined and it can contain any number of nested scopes.

Symbols must be uniquely named in a scope, therefore in big programs it's hard to come up with useful and easy to type names. That's why local and anonymous symbols exists. And grouping certain related symbols into a scope makes sense sometimes too.

Scopes are usually created by `.proc` and `.block` directives, but there are a few other ways. Symbols in a scope can be accessed by using the dot operator, which is applied between the name of the scope and the symbol (e.g. `myconsts.math.pi`).

3.13.1 Regular symbols

Regular symbol names are starting with a letter and containing letters, numbers and underscores. Unicode letters are allowed if the "-a" command line option was used. There's no restriction on the length of symbol names.

Care must be taken to not use duplicate names in the same scope when the symbol is used as a constant. Case sensitivity can be enabled with the "-C" command line option, otherwise symbols are matched case insensitive.

Duplicate names in parent scopes are never a problem, they'll just be “shadowed”. This could be either good by reducing collisions and gives the ability to override “defaults” defined in lower scopes. On the other hand it's possible to mix-up the new symbol with a old one by mistake, which is hard to notice.

A regular symbol is looked up first in the current scope, then in lower scopes until the global scope is reached.

```
f       .block
g        .block
n        nop            ;jump here
         .bend
        .bend

        jsr f.g.n       ;reference from a scope
f.x     = 3             ;create x in scope f with value 3
```

3.13.2 Local symbols

Local symbols have their own scope between two regularly named code symbols and are assigned to the code symbol above them.

Therefore they're easy to reuse without explicit scope declaration directives.

Not all regularly named symbols can be scope boundaries just plain code symbol ones without anything or an opcode after them (no macros!). Symbols defined as procedures, blocks, macros, functions, structs and unions are ignored. Also symbols defined by `.var`, `:=` or `=` don't apply, and there are a few more exceptions, so stick to using plain code labels.

The name must start with an underscore (_), otherwise the same character restrictions apply as for regular symbols. There's no restriction on the length of the name.

Care must be taken to not use the duplicate names in the same scope when the symbol is used as a constant.

A local symbol is only looked up in it's own scope and nowhere else.

```
incr    inc ac
        bne _skip
        inc ac+1
_skip   rts

decr    lda ac
        bne _skip
        dec ac+1
_skip   dec ac          ;symbol reused here
        jmp incr._skip  ;this works too, but is not advised
```

3.13.3 Anonymous symbols

Anonymous symbols don't have a unique name and are always called as a single plus or minus sign. They are also called as forward (+) and backward (-) references.

When referencing them “-” means the first backward, “--” means the second backwards and so on. It's the same for forward, but with “+”. In expressions it may be necessary to put them into brackets.

```
        ldy #4
-       ldx #0
-       txa
        cmp #3
        bcc +
        adc #44
+       sta $400,x
        inx
        bne -
```

```
        dey
        bne --
```

Excessive nesting or long distance references create poorly readable code. It's also very easy to copy-paste a few lines of code with these references into a code fragment already containing similar references. The result is usually a long debugging session to find out what went wrong.

These references are also useful in segments, but this can create a nice trap when segments are copied into the code with their internal references.

```
        bne +
        #somemakro      ;let's hope that this segment does
+       nop             ;not contain forward references...
```

A anonymous symbols are looked up first in the current scope, then in lower scopes until the global scope is reached.

3.13.4 Constant and re-definable symbols

Constant symbols can be created with the equal sign. These are not re-definable. Forward referencing of them is allowed as they retain the objects over compilation passes.

Symbols in front of code or certain assembler directives are created as constant symbols too. They are binded to the object following them.

Re-definable symbols can be created by the `.var` directive or `:=` construct. These are also called as variables as they don't carry their content over from the previous pass. Therefore it's not possible to use them before their definition.

```
border  = $d020         ;a constant
        inc border      ;inc $d020
variabl .var 1          ;a variable
var2    := 1            ;another variable
        .rept 10
        .byte variabl
variabl .var variabl+1  ;increment it
        .next
```

3.13.5 The star label

The “*” symbol denotes the current program counter value. When accessed it's value is the program counter at the beginning of the line. Assigning to it changes the program counter and the compiling offset.

3.14 Built-in functions

Builting functions are assigned to the symbols listed below. If you reuse these symbols in a scope for other purposes then they become inaccessible, or can perform a different function.

Built-in functions can be assigned to symbols (e.g. `sinus = sin`), and the new name can be used as the original function. They can even be passed as parameters to functions.

3.14.1 Mathematical functions

floor(<expression>)
: Round down. E.g. `floor(-4.8)` is `-5.0`

round(<expression>)
: Round to nearest away from zero. E.g. `round(4.8)` is `5.0`

ceil(<expression>**)**
: Round up. E.g. ceil(1.1) is 2.0

trunc(<expression>**)**
: Round down towards zero. E.g. trunc(-1.9) is -1

frac(<expression>**)**
: Fractional part. E.g. frac(1.1) is 0.1

sqrt(<expression>**)**
: Square root. E.g. sqrt(16.0) is 4.0

cbrt(<expression>**)**
: Cube root. E.g. cbrt(27.0) is 3.0

log10(<expression>**)**
: Common logarithm. E.g. log10(100.0) is 2.0

log(<expression>**)**
: Natural logarithm. E.g. log(1) is 0.0

exp(<expression>**)**
: Exponential. E.g. exp(0) is 1.0

pow(<expression a>, <expression b>**)**
: A raised to power of B. E.g. pow(2.0, 3.0) is 8.0

sin(<expression>**)**
: Sine. E.g. sin(0.0) is 0.0

asin(<expression>**)**
: Arc sine. E.g. asin(0.0) is 0.0

sinh(<expression>**)**
: Hyperbolic sine. E.g. sinh(0.0) is 0.0

cos(<expression>**)**
: Cosine. E.g. cos(0.0) is 1.0

acos(<expression>**)**
: Arc cosine. E.g. acos(1.0) is 0.0

cosh(<expression>**)**
: Hyperbolic cosine. E.g. cosh(0.0) is 1.0

tan(<expression>**)**
: Tangent. E.g. tan(0.0) is 0.0

atan(<expression>**)**
: Arc tangent. E.g. atan(0.0) is 0.0

tanh(<expression>**)**
: Hyperbolic tangent. E.g. tanh(0.0) is 0.0

rad(<expression>**)**
: Degrees to radian. E.g. rad(0.0) is 0.0

deg(<expression>**)**
: Radian to degrees. E.g. deg(0.0) is 0.0

hypot(<expression y>, <expression x>**)**
: Polar distance. E.g. hypot(4.0, 3.0) is 5.0

atan2(<expression y>, <expression x>**)**
: Polar angle in −pi to +pi range. E.g. atan2(0.0, 3.0) is 0.0

abs(<expression>**)**
: Absolute value. E.g. abs(-1) is 1

sign(<expression>**)**
: Returns the sign of value as −1, 0 or 1 for negative, zero and positive. E.g. sign(-5) is -1

3.14.2 Other functions

all(<expression>)

Return truth for various definitions of “all”.

all bits set or no bits at all	`all($f)` is `true`
all characters non-zero or empty string	`all("c")` is `true`
all bytes non-zero or no bytes	`all(b"c")` is `true`
all elements true or empty list	`all([1, 1, 0])` is `false`

Table 13: All function

any(<expression>)

Return truth for various definitions of “any”.

at least one bit set	`any(~$f)` is `false`
at least one non-zero character	`any("c")` is `true`
at least one non-zero byte	`any(b"c")` is `true`
at least one true element	`any([1, 1, 0])` is `true`

Table 14: Any function

format(<string expression>[, <expression>, …])

Create string from values according to a format string.

The `format` function converts a list of values into a character string. The converted values are inserted in place of the `%` sign. Optional conversion flags and minimum field length may follow, before the conversion type character. These flags can be used:

`#`	alternate form ($a, %10, 10.)
`*`	width/precision from list
`.`	precision
`0`	pad with zeros
`-`	left adjusted (default right)
	blank when positive or minus sign
`+`	sign even if positive

Table 15: Formatting flags

The following conversion types are implemented:

`a A`	hexadecimal floating point (uppercase)
`b`	binary
`c`	unicode character
`d`	decimal
`e E`	exponential float (uppercase)
`f F`	floating point (uppercase)
`g G`	exponential/floating point
`s`	string
`r`	representation
`x X`	hexadecimal (uppercase)
`%`	percent sign

Table 16: Formatting conversion types

```
        .text format("%#04x bytes left", 1000); $03e8 bytes left
```

len(<expression>)

Returns the number of elements.

bit string	length in bits	`len($034)` is `12`
character string	number of characters	`len("abc")` is `3`
byte string	number of bytes	`len(b"abc")` is `3`

Table 17: Length of various types

tuple, list	number of elements	`len([1, 2, 3])` is `3`
dictionary	number of elements	`len({1:2, 3:4])` is `2`
code	number of elements	`len(label)`

random([<expression>, …]**)**

Returns a pseudo random number.

The sequence does not change across compilations and is the same every time. Different sequences can be generated by seeding.

floating point number 0.0 <= x < 1.0	`random()`
integer in range of 0 <= x < e	`random(e)`
integer in range of s <= x < e	`random(s, a)`
integer in range of s <= x < e, step t	`random(s, a, t)`

Table 18: Random function invocation types

```
        .seed 1234      ; default is boring, seed the generator
        .byte random(256); a pseudo random byte (0..255)
```

range(<expression>[, <expression>, …]**)**

Returns a list of integers in a range, with optional stepping.

integers from 0 to e−1	`range(e)`
integers from s to e−1	`range(s, a)`
integers from s to e (not including e), step t	`range(s, a, t)`

Table 19: Range function invocation types

```
        .byte range(16) ; 0, 1, ..., 14, 15
        .char range(-5, 6); -5, -4, ..., 4, 5
mylist  = range(10, 0, -2); [10, 8, 6, 4, 2]
```

repr(<expression>**)**

Returns a string representation of value.

```
        .warn repr(var) ; pretty print value, for debugging
```

size(<expression>**)**

Returns the size of code, structure or union in bytes.

```
        ldx #size(var) ; size to x
```

3.15 Expressions

3.15.1 Operators

The following operators are available. Not all are defined for all types of arguments and their meaning might slightly vary depending on the type.

`-`	negative	`+`	positive
`!`	not	`~`	invert
`*`	convert to arguments	`^`	decimal string

Table 20: Unary operators

`+`	add	`-`	subtract
`*`	multiply	`/`	divide
`%`	modulo	`**`	raise to power
`|`	binary or	`^`	binary xor
`&`	binary and	`<<`	shift left

Table 21: Binary operators

`>>`	shift right	`.`	member
`..`	concat	`x`	repeat
`in`	contains		

There's a ternary operator (`?:`) which gives the second value if the first is true or the third if the first is false.

Parenthesis (`( )`) can be used to override operator precedence. Don't forget that they also denote indirect addressing mode for certain opcodes.

```
        lda #(4+2)*3
```

3.15.2 Comparison operators

Traditional comparison operators give false or true depending on the result.

The compare operator (`<=>`) gives −1 for less, 0 for equal and 1 for more.

`<=>`	compare		
`==`	equals	`!=`	not equal
`<`	less than	`>=`	more than or equals
`>`	more than	`<=`	less than or equals

Table 22: Comparison operators

3.15.3 Bit string extraction operators

These unary operators extract 8 or 16 bits as a bit string from various types of operands.

`<`	lower byte	`>`	higher byte
`<>`	lower word	`` >` ``	higher word
`><`	lower byte swapped word	`` ` ``	bank byte

Table 23: Bit string extraction operators

```
        lda #<label
        ldy #>label
        jsr $ab1e

        ldx #<>source   ; word extraction
        ldy #<>dest
        lda #size(source)-1
        mvn #`source, #`dest; bank extraction
```

3.15.4 Conditional operators

Boolean conditional operators give false or true or one of the operands as the result. True is defined as a non-zero number, anything else is false.

<table>
<tr><td><code>x || y</code></td><td>if x is true then x otherwise y</td></tr>
<tr><td><code>x ^^ y</code></td><td>if both false or true then false otherwise x || y</td></tr>
<tr><td><code>x && y</code></td><td>if x is true then y otherwise x</td></tr>
<tr><td><code>!x</code></td><td>if x is true then false otherwise true</td></tr>
<tr><td><code>!!x</code></td><td>if x is true then true otherwise false</td></tr>
<tr><td><code>c ? x : y</code></td><td>if c is true then x otherwise y</td></tr>
</table>

Table 24: Logical and conditional operators

```
;Silly example for 1=>"simple", 2=>"advanced", else "normal"
        .text MODE == 1 && "simple" || MODE == 2 && "advanced" || "normal"
        .text MODE == 1 ? "simple" : MODE == 2 ? "advanced" : "normal"
```

Please note that these are not short circuiting operations and both sides are calculated even if thrown away later.

3.15.5 Address length forcing

Special addressing length forcing operators in front of an expression can be used to make sure the expected addressing mode is used. Only applicable when used directly with instructions.

`@b`	to force 8 bit address
`@w`	to force 16 bit address
`@l`	to force 24 bit address (65816)

Table 25: Address size forcing

```
        lda @w$0000
```

3.15.6 Compound assignment

These assignment operators are shorthands for common `.var` directive use.

With the exception of `:=` the variables updated must be defined beforehand. As with `.var` they can't update constants, only variables.

`+=`	add	`-=`	subtract
`*=`	multiply	`/=`	divide
`%=`	modulo	`**=`	raise to power
`|=`	binary or	`^=`	binary xor
`&=`	binary and	`<<=`	shift left
`>>=`	shift right	`..=`	concat
`x=`	repeat	`:=`	assign

Table 26: Compound assignments

```
v       := 1            ; same as 'v .var 1'
v       += 1            ; same as 'v .var v + 1'
```

4 Compiler directives

4.1 Controlling the compile offset and program counter

Two counters are used while assembling.

The compile offset is where the data and code ends up in memory (or in image file).

The program counter is what labels get set to and what the special star label refers to. It wraps when the border of a 64 KiB program bank is crossed. The actual program bank is not incremented, just like on a real processor.

Normally both are the same (code is compiled to the location it runs from) but it does not need to be.

`*= <expression>`
: The compile offset is adjusted so that the program counter will match the requested address in the expression.

```
;Offset PC       Bytes          Disassembly      Source
                                                 *        = $0800
>0800                                                     .byte
                                                          .logical $1000
>0800   1000                                              .byte
```

```
                                                *       = $1200
>0a00   1200                                            .byte
                                                        .here
>0a00                                                   .byte
```

.offs <expression>
: Add an offset to the compile offset (create a gap). The program counter stays the same as before.

```
;Offset PC        Bytes          Disassembly      Source
                                                *       = $1000
.1000                            nop                    .byte
                                                        .offs 100
.1064   1000                     nop                    .byte
```

.logical <expression>
.here
: Changes the program counter only, the compile offset is not changed. Used for code copied to it's proper location at runtime. Can be nested of course.

```
;Offset PC        Bytes          Disassembly      Source
                                                *       = $1000
                                                        .logical $300
.1000   0300      a9 80          lda #$80         drive lda #$80
.1002   0302      85 00          sta $00                sta $00
.1004   0304      4c 00 03       jmp $0300              jmp drive
                                                        .here
```

.align <expression>[, <fill>]
: Align code to a dividable program counter address by inserting uninitialized memory or repeated bytes.

```
;Offset PC        Bytes          Disassembly      Source
                                                *       = $ffc
>0ffc                                                   .align $100
.1000             ee 19 d0       inc $d019        irq   inc $d019
>1003             ea                                    .align 4, $ea
.1004             69 01          adc #$01         loop  adc #1
```

4.2 Dumping data

4.2.1 Storing numeric values

Multi byte numeric data is stored in the little-endian order, which is the natural byte order for 65xx processors. Numeric ranges are enforced depending on the directives used.

When using lists or tuples their content will be used one by one. Uninitialized data ("`?`") creates holes of different sizes. Character string constants are converted using the current encoding.

Please note that multi character strings usually don't fit into 8 bits and therefore the `.byte` directive is not appropriate for them. Use `.text` instead which accepts strings of any length.

.byte <expression>[, <expression>, …]
: Create bytes from 8 bit unsigned constants (0–255)

.char <expression>[, <expression>, …]
: Create bytes from 8 bit signed constants (−128–127)

```
        .byte 255       ; $ff
```

```
        .byte "a"       ; single character
        .byte ?         ; reserve 1 byte of space
        .char -3        ; $fd
;Store 4.4 signed fixed point constants
        .byte (-3.5, 3.25, 3.125) * 1p4

;Compact computed jumps using self modifying code
        lda jumps,x
        sta smod+1
smod    bne *
jumps   .char (routine1, routine2)-smod-2 ;Routines nearby (-128-127 bytes)
```

.word `<expression>[, <expression>, …]`
: Create bytes from 16 bit unsigned constants (0–65535)

.int `<expression>[, <expression>, …]`
: Create bytes from 16 bit signed constants (−32768–32767)

```
        .word $2342, $4555; $42 $23 $55 $45
        .word ?         ; reserve 2 bytes of space
        .int -533, 4433 ; $eb $fd $51 $11
;Store 8.8 signed fixed point constants
        .int (-3.5, 3.25, 3.125) * 1p8

;Computed jumps with jump table (bank zero or non-65816)
        lda jumps,x
        sta ind
        lda jumps+1,x
        sta ind+1
        jmp (ind)
jumps   .word routine1, routine2; but better use .addr instead
```

.addr `<expression>[, <expression>, …]`
: Create 16 bit address constants for addresses (in current program bank)

.rta `<expression>[, <expression>, …]`
: Create 16 bit return address constants for addresses (in current program bank)

```
;Computed jumps with jump table (65816, current bank)
*       = $12000
        jmp (jumps,x)
jumps   .addr $12050, routine1, routine2

;Computed jumps by using stack (current bank)
*       = $103000
        lda rets+1,x
        pha
        lda rets,x
        pha
        rts
rets    .rta $10f000, routine1, routine2
```

.long `<expression>[, <expression>, …]`
: Create bytes from 24 bit unsigned constants (0–16777215)

.lint `<expression>[, <expression>, …]`
: Create bytes from 24 bit signed constants (−8388608–8388607)

```
        .long $123456   ; $56 $34 $12
        .long ?         ; reserve 3 bytes of space
```

```
        .lint -533, 4433; $eb $fd $ff $51 $11 $00
;Store 8.16 signed fixed point constants
        .lint (-3.44, 3.4, 3.52) * 1p16

;Computed long jumps with jump table (65816)
        lda jumps,x
        sta ind
        lda jumps+1,x
        sta ind+1
        lda jumps+2,x
        sta ind+2
        jmp [ind]
jumps   .long routine1, routine2
```

.dword <expression>[, <expression>, …]
: Create bytes from 32 bit constants (0–4294967295)

.dint <expression>[, <expression>, …]
: Create bytes from 32 bit signed constants (−2147483648–2147483647)

```
        .dword $12345678; $78 $56 $34 $12
        .dword ?        ; reserve 4 bytes of space
        .dint -411469219; $5d $7a $79 $e7
;Store 16.16 signed fixed point constants
        .dint (-3.44, 3.4, 3.52) * 1p16
```

4.2.2 Storing string values

The following directives store strings of characters, bytes or bits as bytes. Small numeric constants can be mixed in to represent single byte control characters.

When using lists or tuples their content will be used one by one. Uninitialized data ("?") creates byte sized holes. Character string constants are converted using the current encoding.

.text <expression>[, <expression>, …]
: Assemble strings without conversion into bytes.

```
        .text "oeU"      ; text, "" means $22
        .text 'oeU'      ; text, '' means $27
        .text 23, $33    ; bytes
        .text $0a0d      ; $0d, $0a, little endian!
        .text %00011111  ; more bytes
        .text ^OEU       ; the decimal value as string (^23 is $32,$33)
```

.fill <length>[, <fill>]
: Skip bytes (using uninitialized data), or fill with repeated bytes.

```
        .fill $100       ;no fill, just reserve $100 bytes
        .fill $4000, 0   ;16384 bytes of 0
        .fill 8000, [$55, $aa];8000 bytes of alternating $55, $aa
```

.shift <expression>[, <expression>, …]
: Same as `.text`, but the last byte will have the highest bit set. Any byte which already has the most significant bit set will cause an error. The last byte can't be uninitialized or missing of course.

```
        ldx #0
loop    lda txt,x
```

```
        php
        and #$7f
        jsr $ffd2
        inx
        plp
        bpl loop
        rts
txt     .shift "single", 32, "string"
        .text s"first", s"second"
```

.shiftl <expression>[, <expression>, …]

Same as `.text`, but all bytes are shifted to left, and the last byte gets the lowest bit set. Any byte which already has the most significant bit set will cause an error as this is cut off on shifting. The last byte can't be uninitialized or missing of course.

```
        ldx #0
loop    lda txt,x
        lsr
        sta $400,x      ;screen memory
        inx
        bcc loop
        rts
        .enc screen
txt     .shiftl "single", 32, "string"
        .text l"first", l"second"
        .enc none
```

.null <expression>[, <expression>, …]

Same as `.text`, but adds a zero byte to the end. An existing zero byte is an error as it'd cause a false end marker.

```
        lda #<txt
        ldy #>txt
        jsr $ab1e
txt     .null "single", 32, "string"
        .text n"first", n"second"
```

.ptext <expression>[, <expression>, …]

Same as `.text`, but prepend the number of bytes in front of the string (pascal style string). Therefore it can't do more than 255 bytes.

```
        lda #<txt
        ldx #>txt
        jsr print
        rts

print   sta $fb
        stx $fc
        ldy #0
        lda ($fb),y
        beq null
        tax
-       iny
        lda ($fb),y
        jsr $ffd2
        dex
        bne -
null    rts
```

```
txt     .ptext "single", 32, "string"
        .text p"first", p"second"
```

4.3 Text encoding

64tass supports sources written in UTF-8, UTF-16 (be/le) and RAW 8 bit encoding. To take advantage of this capability custom encodings can be defined to map Unicode characters to 8 bit values in strings.

.enc <name>
: Selects text encoding, predefined encodings are “none” and “screen” (screen code), anything else is user defined. All user encodings start without any character or escape definitions, add some as required.

```
        .enc screen     ;screen code mode
        .text "text with screen codes"
        cmp #"u"        ;compare screen code
        .enc none       ;normal mode again
        cmp #"u"        ;compare ASCII
```

.cdef <start>, <end>, <coded> [, <start>, <end>, <coded>, …]
.cdef "<start><end>", <coded> [, "<start><end>", <coded>, …]
: Assigns characters in a range to single bytes.

  This is a simple single character to byte translation definition. It is applied to a range as characters and bytes are usually assigned sequentially. The start and end positions are Unicode character codes either by numbers or by typing them. Overlapping ranges are not allowed.

.edef "<escapetext>", <value> [, "<escapetext>", <value>, …]
: Assigns strings to byte sequences as a translated value.

  When these substrings are found in a text they are replaced by bytes defined here. When strings with common prefixes are used the longest match wins. Useful for defining non-typeable control code aliases, or as a simple tokenizer.

```
        .enc petscii    ;define an ascii->petscii encoding
        .cdef " @", 32  ;characters
        .cdef "AZ", $c1
        .cdef "az", $41
        .cdef "[[", $5b
        .cdef "££", $5c
        .cdef "]]", $5d
        .cdef "ππ", $5e
        .cdef $2190, $2190, $1f;left arrow

        .edef "\n", 13  ;one byte control codes
        .edef "{clr}", 147
        .edef "{crlf}", [13, 10];two byte control code
        .edef "<nothing>", [];replace with no bytes

        .text "{clr}Text in PETSCII\n"
```

4.4 Structured data

Structures and unions can be defined to create complex data types. The offset of fields are available by using the definition's name. The fields themselves by using the instance name.

The initialization method is very similar to macro parameters, the difference is that unset parameters always return uninitialized data ("?") instead of an error.

4.4.1 Structure

Structures are for organizing sequential data, so the length of a structure is the sum of lengths of all items.

.struct [<name>][=<default>]][, [<name>][=<default>] …]
.ends [<result>][, <result> …]
  Structure definition, with named parameters and default values

.dstruct <name>[, <initialization values>]
.<name> [<initialization values>]
  Create instance of structure with initialization values

```
        .struct         ;anonymous structure
x       .byte 0         ;labels are visible
y       .byte 0         ;content compiled here
        .ends           ;useful inside unions

nn_s    .struct col, row;named structure
x       .byte \col      ;labels are not visible
y       .byte \row      ;no content is compiled here
        .ends           ;it's just a definition

nn      .dstruct nn_s, 1, 2;structure instance, content here

        lda nn.x        ;direct field access
        ldy #nn_s.x     ;get offset of field
        lda nn,y        ;and use it indirectly
```

4.4.2 Union

Unions can be used for overlapping data as the compile offset and program counter remains the same on each line. Therefore the length of a union is the length of it's longest item.

.union [<name>][=<default>]][, [<name>][=<default>] …]
.endu
  Union definition, with named parameters and default values

.dunion <name>[, <initialization values>]
.<name> [<initialization values>]
  Create instance of union with initialization values

```
        .union          ;anonymous union
x       .byte 0         ;labels are visible
y       .word 0         ;content compiled here
        .endu

nn_u    .union          ;named union
x       .byte ?         ;labels are not visible
y       .word \1        ;no content is compiled here
        .endu           ;it's just a definition

nn      .dunion nn_u, 1 ;union instance here

        lda nn.x        ;direct field access
        ldy #nn_u.x     ;get offset of field
        lda nn,y        ;and use it indirectly
```

4.4.3 Combined use of structures and unions

The example below shows how to define structure to a binary include.

```
        .union
        .binary "pic.drp", 2
        .struct
color   .fill 1024
screen  .fill 1024
bitmap  .fill 8000
backg   .byte ?
        .ends
        .endu
```

Anonymous structures and unions in combination with sections are useful for overlapping memory assignment. The example below shares zeropage allocations for two separate parts of a bigger program. The common subroutine variables are assigned after in the “zp” section.

```
*       = $02
        .union          ;spare some memory
         .struct
          .dsection zp1 ;declare zp1 section
         .ends
         .struct
          .dsection zp2 ;declare zp2 section
         .ends
        .endu
        .dsection zp    ;declare zp section
```

4.5 Macros

Macros can be used to reduce typing of frequently used source lines. Each invocation is a copy of the macro's content with parameter references replaced by the parameter texts.

.segment [<name>][=<default>]][, [<name>][=<default>] …]
.endm [<result>][, <result> …]
: Copies the code segment as it is, so symbols can be used from outside, but this also means multiple use will result in double defines unless anonymous labels are used.

.macro [<name>][=<default>]][, [<name>][=<default>] …]
.endm [<result>][, <result> …]
: The code is enclosed in it's own block so symbols inside are non-accessible, unless a label is prefixed at the place of use, then local labels can be accessed through that label.

#<name> [<param>][[,][<param>] …]
.<name> [<param>][[,][<param>] …]
: Invoke the macro after “#” or “.” with the parameters. Normally the name of the macro is used, but it can be any expression.

```
;A simple macro
copy    .macro
        ldx #size(\1)
lp      lda \1,x
        sta \2,x
        dex
        bpl lp
        .endm
```

```
        #copy label, $500

;Use macro as an assembler directive
lohi    .macro
lo      .byte <(\@)
hi      .byte >(\@)
        .endm

var     .lohi 1234, 5678

        lda var.lo,y
        ldx var.hi,y
```

4.5.1 Parameter references

The first 9 parameters can be referenced by “`\1`”–“`\9`”. The entire parameter list including separators is “`\@`”.

```
name    .macro
        lda #\1         ;first parameter 23+1
        .endm

        #name 23+1      ;call macro
```

Parameters can be named, and it's possible to set a default value after an equal sign which is used as a replacement when the parameter is missing.

These named parameters can be referenced by `\name` or `\{name}`. Names must match completely, if unsure use the quoted name reference syntax.

```
name    .macro first, b=2, , last
        lda #\first     ;first parameter
        lda #\b         ;second parameter
        lda #\3         ;third parameter
        lda #\last      ;fourth parameter
        .endm

        #name 1, , 3, 4 ;call macro
```

4.5.2 Text references

In the original turbo assembler normal references are passed by value and can only appear in place of one. Text references on the other hand can appear everywhere and will work in place of e.g. quoted text or opcodes and labels. The first 9 parameters can be referenced as text by `@1`–`@9`.

```
name    .macro
        jsr print
        .null "Hello @1!";first parameter
        .endm

        #name "wth?"    ;call macro
```

4.6 Custom functions

Beyond the built-in functions mentioned earlier it's possible to define custom ones for frequently used calculations.

.function <name>[=<default>]][, <name>[=<default>] …][, *<name>]
.endf [<result>][, <result> …]
    Defines a user function

#<name> [<param>][[,][<param>] …]
.<name> [<param>][[,][<param>] …]
<name> [<param>][[,][<param>] …]
    Invoke a function like a macro, directive or pseudo instruction.

Parameters are assigned to constant symbols in the function scope on invocation. The default values are calculated at function definition time only, and these values are used at invocation time when a parameter is missing.

Extra parameters are not accepted, unless the last parameter symbol is preceded with a star, in this case these parameters are collected into a tuple. Multiple values are returned are also returned as tuple.

Functions can span multiple lines but unlike macros they can't create new code. Only those external variables and functions are available which were accessible at the place of definition, but not those at the place of invocation.

```
wpack   .function a, b=0
        .endf a+b*256

        .word wpack(1), wpack(2, 3)
```

If a function is used as macro, directive or pseudo instruction and there's a label in front then the returned value is assigned to it. If nothing is returned then it's used as regular label. Of course when used like this it can create code and access local variables.

```
mva     .function s, d
        lda s
        sta d
        .endf

        mva #1, label
```

4.7 Conditional assembly

To prevent parts of source from compiling conditional constructs can be used. This is useful when multiple slightly different versions needs to be compiled from the same source.

4.7.1 If, else if, else

.if <expression>
    Compile, if result is true (not zero)

.elsif <expression>
    Compile if the previous conditions were all skipped and the result is true (not zero)

.else
    Compile if the previous conditions were all skipped

.fi
.endif
    End of conditional compile

.ifne <value>
    Compile, if value is not zero (or true)

.ifeq <value>
    Compile, if value is zero (or false)

`.ifpl <value>`
: Compile, if value is greater or equal zero

`.ifmi <value>`
: Compile, if value is less than zero

The `.ifne`, `.ifeq`, `.ifpl` and `.ifmi` directives exists for compatibility only, in practice it's better to use comparison operators instead.

```
        .if wait==2     ;2 cycles
        nop
        .elsif wait==3  ;3 cycles
        bit $ea
        .elsif wait==4  ;4 cycles
        bit $eaea
        .else           ;else 5 cycles
        inc $2
        .fi
```

4.7.2 Switch, case, default

Similar to the `.if`/`.elsif`/`.else`/`.fi` construct, but the compared value needs to be written only once in the switch statement.

`.switch <expression>`
: Evaluate expression and remember it

`.case <expression>[, <expression> …]`
: Compile if the previous conditions were all skipped and one of the values equals

`.default`
: Compile if the previous conditions were all skipped

`.endswitch`
: End of conditional compile

```
        .switch wait
        .case 2         ;2 cycles
        nop
        .case 3         ;3 cycles
        bit $ea
        .case 4         ;4 cycles
        bit $eaea
        .default        ;else 5 cycles
        inc $2
        .endswitch
```

4.8 Repetitions

`.for [<variable>=<expression>], [<condition expression>], [<variable>=<expression>]`
`.next`
: Loop while the condition is true. If there's no condition then it's an infinite loop and `.break` must be used to terminate it.

```
        ldx #0
        lda #32
lp      .for ue = $400, ue < $800, ue = ue + $100
        sta ue,x
        .next
        dex
```

```
        bne lp
```

.rept <expression>
.next
Repeat by expression number of times.

```
        .rept 100
        nop
        .next
```

.break
Exit current loop immediately

.continue
Continue current loop's next iteration

.lbl
Creates a special jump label that can be referenced by `.goto`

.goto <labelname>
Causes assembler to continue assembling from the jump label. No forward references of course, handle with care. Typically used in classic TASM sources for creating loops.

```
i       .var 100
loop    .lbl
        nop
i       .var i - 1
        .ifne i
        .goto loop      ;generates 100 nops
        .fi
```

4.9 Including files

Longer sources are usually separated into multiple files for easier handling. Precomputed binary data can also be included directly without converting it into source code first.

Search path is relative to the location of current source file. If it's not found there the include search path is consulted for further possible locations.

To make your sources portable please always use forward slashes (/) as a directory separator and use lower/uppercase consistently in filenames!

.include <filename>
Include source file here.

.binclude <filename>
Include source file here in it's local block. If the directive is prefixed with a label then all labels are local and are accessible through that label only, otherwise not reachable at all.

```

        .include "macros.asm"       ;include macros
menu    .binclude "menu.asm"        ;include in a block
        jmp menu.start
```

.binary <filename>[, <offset>[, <length>]]
Include raw binary data from file. By using offset and length it's possible to break out chunks of data from a file separately, like bitmap and colors for example.

```
        .binary "stuffz.bin"        ;simple include, all bytes
        .binary "stuffz.bin", 2     ;skip start address
        .binary "stuffz.bin", 2, 1000;skip start address, 1000 bytes max
```

```
*       = $1000                     ;load music to $1000 and
        .binary "music.sid", $7e    ;strip SID header
```

4.10 Scopes

Scopes may contain symbols or other scopes nested. They are useful to avoid symbol clashes as the same symbol name can repeated as long as it's in a different scope.

In nested scopes the symbol lookup starts from the local scope and goes in the direction of the global scope. This means that local variables will “shadow” global one with the same name.

.proc
.pend

Procedure start and end of procedure.

If it's label is not used then the code won't be compiled at all. This is very useful to avoid a lot of `.if` blocks to exclude unused sections of code.

All labels inside are local enclosed in a scope and are accessible through the prefixed label. Useful for building libraries.

```
ize     .proc
        nop
cucc    nop
        .pend

        jsr ize
        jmp ize.cucc
```

.block
.bend

Block start and block end.

All labels inside a block are local enclosed in a scope. If prefixed with a label local variables are accessible through that label using the dot notation, otherwise not at all.

```
        .block
        inc count + 1
count   ldx #0
        .bend
```

.weak
.endweak

Weak symbol area

Any symbols defined inside can be overriden by “stronger” symbols in the same scope from outside. Can be nested as necessary.

This gives the possibility of giving default values for symbols which might not always exist without resorting to `.ifdef`/`.ifndef` or similar directives in other assemblers.

```
symbol  = 1             ;stronger symbol than the one below
        .weak
symbol  = 0             ;default value if the one above does not exists
        .endweak
        .if symbol      ;almost like an .ifdef ;)
```

Other use of weak symbols might be in included libraries to change default values or replace stub functions and data structures.

If these stubs are defined using .proc/.pend then their default implementations will not even exists in the output at all when a stronger symbol overrides them.

Multiple definition of a symbol with the same "strength" in the same scope is of course not allowed and it results in double definition error.

Please note that .ifdef/.ifndef directives are left out from 64tass for of technical reasons, so don't wait for them to appear anytime soon.

4.11 Sections

Sections can be used to collect data or code into separate memory areas without moving source code lines around. This is achieved by having separate compile offset and program counters for each defined section.

.section <name>
.send [<name>]
Defines a section fragment. The name at .send must match but it's optional.

.dsection <name>
Collect the section fragments here.

All .section fragments are compiled to the memory area allocated by the .dsection directive. Compilation happens as the code appears, this directive only assigns enough space to hold all the content in the section fragments.

The space used by section fragments is calculated from the difference of starting compile offset and the maximum compile offset reached. It is possible to manipulate the compile offset in fragments, but putting code before the start of .dsection is not allowed.

```
*       = $02
        .dsection zp   ;declare zeropage section
        .cerror * > $30, "Too many zeropage variables"

*       = $334
        .dsection bss   ;declare uninitialized variable section
        .cerror * > $400, "Too many variables"

*       = $0801
        .dsection code   ;declare code section
        .cerror * > $1000, "Program too long!"

*       = $1000
        .dsection data   ;declare data section
        .cerror * > $2000, "Data too long!"
;--------------------
        .section code
        .word ss, 2005
        .null $9e, ^start
ss      .word 0

start   sei
        .section zp     ;declare some new zeropage variables
p2      .word ?         ;a pointer
        .send zp
        .section bss    ;new variables
buffer  .fill 10        ;temporary area
        .send bss

        lda (p2),y
        lda #<label
```

```
        ldy #>label
        jsr print

        .section data   ;some data
label   .null "message"
        .send data

        jmp error
        .section zp     ;declare some more zeropage variables
p3      .word ?         ;a pointer
        .send zp
        .send code
```

The compiled code will look like:

```
>0801    0b 08 d5 07                        .word ss, 2005
>0805    9e 32 30 36 31 00                  .null $9e, ^start
>080b    00 00                      ss      .word 0

.080d    78                         start   sei

>0002                               p2      .word ?         ;a pointer
>0334                               buffer  .fill 10        ;temporary area

.080e    b1 02                              lda (p2),y
.0810    a9 00                              lda #<label
.0812    a0 10                              ldy #>label
.0814    20 1e ab                           jsr print

>1000    6d 65 73 73 61 67 65 00    label   .null "message"

.0817    4c e2 fc                           jmp error

>0004                               p2      .word ?         ;a pointer
```

Sections can form a hierarchy by nesting a `.dsection` into another section. The section names must only be unique within a section but can be reused otherwise. Parent section names are visible for children, siblings can be reached through parents.

In the following example the included sources don't have to know which “code” and “data” sections they use, while the “bss” section is shared for all banks.

```
;First 8K bank at the beginning, PC at $8000
*       = $0000
        .logical $8000
        .dsection bank1
        .cerror * > $a000, "Bank1 too long"
        .here

bank1   .block          ;Make all symbols local
        .section bank1
        .dsection code  ;Code and data sections in bank1
        .dsection data
        .section code   ;Pre-open code section
        .include "code.asm"; see below
        .include "iter.asm"
        .send code
        .send bank1
        .bend
```

```
;Second 8K bank at $2000, PC at $8000
*       = $2000
        .logical $8000
        .dsection bank2
        .cerror * > $a000, "Bank2 too long"
        .here

bank2   .block          ;Make all symbols local
        .section bank2
        .dsection code  ;Code and data sections in bank2
        .dsection data
        .section code   ;Pre-open code section
        .include "scr.asm"
        .send code
        .send bank2
        .bend

;Common data, avoid initialized variables here!
*       = $c000
        .dsection bss
        .cerror * > $d000, "Too much common data"
;------------- The following is in "code.asm"
code    sei

        .section bss   ;Common data section
buffer  .fill 10
        .send bss

        .section data  ;Data section (in bank1)
routine .word print
        .send bss
```

4.12 65816 related

.as
.al
: Select short (8 bit) or long (16 bit) accumulator immediate constants.

```
        .al
        lda #$4322
```

.xs
.xl
: Select short (8 bit) or long (16 bit) index register immediate constants.

```
        .xl
        ldx #$1000
```

.autsiz
.mansiz
: Select automatic adjustment of immediate constant sizes based on `SEP`/`REP` instructions.

```
        .autsiz
        rep #$10        ;implicit .xl
        ldx #$1000
```

.databank <expression>

Data bank (absolute) addressing is only used for addresses falling into this 64 KiB bank. The default is 0, which means addresses in bank zero.

When data bank is switched off only data bank indexed (,b) addresses create data bank accessing instructions.

```
        .databank $10   ;data bank at $10xxxx
        lda $101234     ;results in $ad, $34, $12
        .databank ?     ;no data bank
        lda $1234       ;direct page or long addressing
        lda $1234,b     ;results in $ad, $34, $12
```

.dpage <expression>
: Direct (zero) page addressing is only used for addresses falling into a specific 256 byte address range. The default is 0, which is the first page of bank zero.

When direct page is switched off only the direct page indexed (,d) addresses create direct page accessing instructions.

```
        .dpage $400     ;direct page $400-$4ff
        lda $456        ;results in $a5, $56
        .dpage ?        ;no direct page
        lda $56         ;data bank or long addressing
        lda $56,d       ;results in $a5, $56
```

4.13 Controlling errors

.page
.endp
: Gives an error on page boundary crossing, e.g. for timing sensitive code.

```
        .page
table   .byte 0, 1, 2, 3, 4, 5, 6, 7
        .endp
```

.option allow_branch_across_page
: Switches error generation on page boundary crossing during relative branch. Such a condition on 6502 adds 1 extra cycle to the execution time, which can ruin the timing of a carefully cycle counted code.

```
        .option allow_branch_across_page = 0
        ldx #3          ;now this will execute in
-       dex             ;16 cycles for sure
        bne -
        .option allow_branch_across_page = 1
```

.error <message> [, <message>, …]
.cerror <condition>, <message> [, <message>, …]
: Exit with error or conditionally exit with error

```
        .error "Unfinished here..."
        .cerror * > $1200, "Program too long by ", * - $1200, " bytes"
```

.warn <message> [, <message>, …]
.cwarn <condition>, <message> [, <message>, …]
: Display a warning message always or depending on a condition

```
        .warn "FIXME: handle negative values too!"
```

```
        .cwarn * > $1200, "This may not work!"
```

4.14 Target

`.cpu <expression>`
: Selects CPU according to the string argument.

```
        .cpu "6502"     ;standard 65xx
        .cpu "65c02"    ;CMOS 65C02
        .cpu "65ce02"   ;CSG 65CE02
        .cpu "6502i"    ;NMOS 65xx
        .cpu "65816"    ;W65C816
        .cpu "65dtv02"  ;65dtv02
        .cpu "65el02"   ;65el02
        .cpu "r65c02"   ;R65C02
        .cpu "w65c02"   ;W65C02
        .cpu "default"  ;cpu set on commandline
```

4.15 Misc

`.end`
: Terminate assembly. Any content after this directive is ignored.

`.eor <expression>`
: XOR output with a 8 bit value. Useful for reverse screen code text for example, or for silly "encryption".

`.seed <expression>`
: Seed the pseudo random number generator with an unsigned integer of maximum 128 bits, to make the generated numbers less boring.

`.var <expression>`
: Defines a variable identified by the label preceding, which is set to the value of expression or reference of variable.

`.comment`
`.endc`
: Comment block start and comment block end.

```
        .comment
        lda #1          ;this won't be compiled
        sta $d020
        .endc
```

`.assert`
`.check`
: Do not use these, the syntax will change in next version!

4.16 Printer control

`.pron`
`.proff`
: Turn on or off source listing on part of the file.

```
        .proff           ;Don't put filler bytes into listing
*       = $8000
        .fill $2000, $ff ;Pre-fill ROM area
        .pron
*       = $8000
        .word reset, restore
```

```
        .text "CBM80"
reset   cld
```

.hidemac
.showmac
Ignored for compatibility

5 Pseudo instructions

5.1 Aliases

For better code readability `BCC` has an alias named `BLT` (**B**ranch **L**ess **T**han) and `BCS` one named `BGE` (**B**ranch **G**reater **E**qual).

```
        cmp #3
        blt exit        ; less than 3?
```

For similar reasons `ASL` has an alias named `SHL` (**SH**ift **L**eft) and `LSR` one named `SHR` (**SH**ift **R**ight). This naming however is not very common.

The implied variants `LSR`, `ROR`, `ASL` and `ROL` are a shorthand for `LSR A`, `ROR A`, `ASL A` and `ROL A`. Using the implied form is considered poor coding style.

For compatibility `INA` and `DEA` is a shorthand of `INC A` and `DEC A`. Therefore there's no "implied" variants like `INC` or `DEC`. The full form with the accumulator is preferred.

The longer forms of `INC X`, `DEC X`, `INC Y`, `DEC Y`, `INC Z` and `DEC Z` are available for `INX`, `DEX`, `INY`, `DEY`, `INZ` and `DEZ`. For this to work care must be taken to not reuse the "x", "y" and "z" single letter register symbols for other purposes. Same goes for "a" of course.

Load instructions with registers are translated to transfer instructions. For example `LDA X` becomes `TXA`.

Store instructions with registers are translated to transfer instructions, but only if it involves the "s" or "b" registers. For example `STX S` becomes `TXS`.

Many illegal opcodes have aliases for compatibility as there's no standard naming convention.

5.2 Always taken branches

For writing short code there are some special pseudo instructions for always taken branches. These are automatically compiled as relative branches when the jump distance is short enough and as `JMP` or `BRL` when longer.

The names are derived from conditional branches and are: `GEQ`, `GNE`, `GCC`, `GCS`, `GPL`, `GMI`, `GVC`, `GVS`, `GLT` and `GGE`.

There's one more called `GRA` for CPUs supporting `BRA`, which is expanded to `BRL` (if available) or `JMP`.

```
.0000    a9 03        lda #$03      in1     lda #3
.0002    d0 02        bne $0006             gne at          ;branch always
.0004    a9 02        lda #$02      in2     lda #2
.0006    4c 00 10     jmp $1000     at      gne $1000       ;branch further
```

If the branch would skip only one byte then the opposite condition is compiled and only the first byte is emitted. This is now a never executed jump, and the relative distance byte after the opcode is the jumped over byte.

If the branch would not skip anything at all then no code is generated.

```
.0009                                       geq in3         ;zero length "branch"
```

```
.0009    18          clc          in3    clc
.000a    b0          bcs                 gcc at2        ;one byte skip, as bcs
.000b    38          sec          in4    sec            ;sec is skipped!
.000c    20 0f 00    jsr $000f    at2    jsr func
.000f                             func
```

Please note that expressions like `Gxx *+2` or `Gxx *+3` are not allowed as the compiler can't figure out if it has to create no code at all, the 1 byte variant or the 2 byte one. Therefore use normal or anonymous labels defined after the jump instruction when jumping forward!

5.3 Long branches

To avoid branch too long errors the assembler also supports long branches. It can automatically convert conditional relative branches to it's opposite and a `JMP` or `BRL`. This can be enabled on the command line using the “--long-branch” option.

```
.0000    ea             nop                  nop
.0001    b0 03          bcs $0006            bcc $1000      ;long branch (6502)
.0003    4c 00 10       jmp $1000
.0006    1f 17 03       bbr 1,$17,$000c      bbs 1,23,$1000 ;long branch (R65C02)
.0009    4c 00 10       jmp $1000
.000c    d0 04          bne $0012            beq $10000     ;long branch (65816)
.000e    5c 00 00 01    jmp $010000
.0012    30 03          bmi $0017            bpl $1000      ;long branch (65816)
.0014    82 e9 1f       brl $1000
.0017    ea             nop                  nop
```

Please note that forward jump expressions like `Bxx *+130`, `Bxx *+131` and `Bxx *+132` are not allowed as the compiler can't decide between a short/long branch. Of course these destinations can be used, but only with normal or anonymous labels defined after the jump instruction.

In the above example extra `JMP` instructions are emitted for each long branch. This is suboptimal and wasting space if there are several long branches to the same location in close proximity. Therefore the assembler might decide to reuse a `JMP` for more than one long branch to save space.

6 Original turbo assembler compatibility

6.1 How to convert source code for use with 64tass

Currently there are two options, either use “TMPview” by Style to convert the sourcefile directly, or do the following:

- load turbo assembler, start (by `SYS 9*4096` or `SYS 8*4096` depending on version)
- ← then l to load a source file
- ← then w to write a source file in PETSCII format
- convert the result to ASCII using petcat (from the vice package)

The resulting file should then (with the restrictions below) assemble using the following command line:

```
64tass -C -T -a -W -i source.asm -o outfile.prg
```

6.2 Differences to the original turbo ass macro on the C64

64tass is nearly 100% compatible with the original “Turbo Assembler”, and supports most of the features of the original “Turbo Assembler Macro”. The remaining notable differences are listed here.

6.3 Labels

The original turbo assembler uses case sensitive labels, use the `-C, --case-sensitive` option to enable this behaviour.

6.4 Expression evaluation

There are a few differences which can be worked around by the `-T, --tasm-compatible` option. These are:

The original expression parser has no operator precedence, but 64tass has. That means that you will have to fix expressions using braces accordingly, for example `1+2*3` becomes `(1+2)*3`.

The following operators used by the original Turbo Assembler are different:

`.`	bitwise or, now `|`
`:`	bitwise eor, now `^`
`!`	force 16 bit address, now `@w`

Table 27: TASM Operator differences

The default expression evaluation is not limited to 16 bit unsigned numbers anymore.

6.5 Macros

Macro parameters are referenced by "`\1`"–"`\9`" instead of using the pound sign.

Parameters are always copied as text into the macro and not passed by value as the original turbo assembler does, which sometimes may lead to unexpected behaviour. You may need to make use of braces around arguments and/or references to fix this.

6.6 Bugs

Some versions of the original turbo assembler had bugs that are not reproduced by 64tass, you will have to fix the code instead.

In some versions labels used in the first `.block` are globally available. If you get a related error move the respective label out of the `.block`.

7 Command line options

7.1 Output options

`-o` `<filename>`
Place output into <filename>. The default output filename is "a.out". This option changes it.

```
64tass a.asm -o a.prg
```

`no option`
Outputs CBM format binaries

The first 2 bytes are the little endian address of the first valid byte (start address). Overlapping blocks are flattened and uninitialized memory is filled up with zeros. Uninitialized memory before the first and after the last valid bytes are not saved.

Used for C64 binaries.

`-b, --nostart`

Output data only without start address

Overlapping blocks are flattened and uninitialized memory is filled up with zeros. Uninitialized memory before the first and after the last valid bytes are not saved.

Useful for small ROM files.

-f, --flat
Flat address space output mode.

Overlapping blocks are flattened and uninitialized memory is filled up with zeros. Uninitialized memory after the last valid byte is not saved.

Useful for creating huge multi bank ROM files (over 64 KiB). See sections for an example.

-n, --nonlinear
Generate nonlinear output file.

Overlapping blocks are flattened. Blocks are saved in sorted order and uninitialized memory is skipped.

Used for linkers.

```
64tass --nonlinear a.asm
*       = $1000
        lda #2
*       = $2000
        nop
```

$02, $00	little endian length, 2 bytes
$00, $10	little endian start $1000
$a9, $02	code
$01, $00	little endian length, 1 byte
$00, $20	little endian start $2000
$ea	code
$00, $00	end marker (length=0)

Table 28: Result of compilation

-X, --long-address
Use 3 byte address/length for CBM and nonlinear output instead of 2 bytes.

```
64tass --long-address --m65816 a.asm
```

--atari-xex
Generate a Atari XEX output file.

Overlapping blocks are kept, continuing blocks are concatenated. Saving happens in the definition order without sorting, and uninitialized memory is skipped in the output.

```
64tass --atari-xex a.asm
*       = $02e0
        .word start       ;run address
*       = $2000
start   rts
```

$ff, $ff	header, 2 bytes
$e0, $02	little endian start $02e0
$e1, $02	little endian last byte $02e1
$00, $20	start address word

Table 29: Result of compilation

`$00, $20`	little endian start $2000
`$00, $20`	little endian last byte $2000
`$60`	code

--apple2

Generate a Apple II output file (DOS 3.3).

Overlapping blocks are flattened and uninitialized memory is filled up with zeros. Uninitialized memory before the first and after the last valid bytes are not saved.

```
64tass --apple-ii a.asm
*       = $0c00
        rts
```

`$00, $0c`	little endian start $0c00
`$01, $00`	little endian length $0001
`$60`	code

Table 30: Result of compilation

7.2 Operation options

-a, --ascii

Use ASCII/Unicode text encoding instead of raw 8-bit

Normally no conversion takes place, this is for backwards compatibility with a DOS based Turbo Assembler editor, which could create PETSCII files for 6502tass. (including control characters of course)

Using this option will change the default “none” and “screen” encodings to map `'a'-'z'` and `'A'-'Z'` into the correct PETSCII range of `$41-$5A` and `$C1-$DA`, which is more suitable for an ASCII editor. It also adds predefined petcat style PETSCII literals to the default encodings, and enables Unicode letters in symbol names.

For writing sources in UTF-8/UTF-16 encodings this option is required!

```
64tass a.asm

.0000    a9 61          lda #$61        lda #"a"

>0002    31 61 41                       .text "1aA"
>0005    7b 63 6c 65 61 72 7d 74        .text "{clear}text{return}more"
>000e    65 78 74 7b 72 65 74 75
>0016    72 6e 7d 6d 6f 72 65

64tass --ascii a.asm

.0000    a9 41          lda #$41        lda #"a"
>0002    31 41 c1                       .text "1aA"
>0005    93 54 45 58 54 0d 4d 4f        .text "{clear}text{return}more"
>000e    52 45
```

-B, --long-branch

Automatic `BXX *+5 JMP xxx`. Branch too long messages can be annoying sometimes, usually they'll need to be rewritten to `BXX *+5 JMP xxx`. 64tass can do this automatically if this option is used. But `BRA` is not converted.

```
64tass a.asm
*       = $1000
        bcc $1233       ;error...
```

```
64tass a.asm
*       = $1000
        bcs *+5         ;opposite condition
        jmp $1233       ;as simple workaround

64tass --long-branch a.asm
*       = $1000
        bcc $1233       ;no error, automatically converted to the above one.
```

-C, --case-sensitive
: Case sensitive labels. Labels are not case sensitive by default, this option changes that.

```
64tass a.asm
label   nop
Label   nop     ;double defined...

64tass --case-sensitive a.asm
label   nop
Label   nop     ;Ok, it's a different label...
```

-D <label>=<value>
: Define <label> to <value>. Defines a label to a value. Same syntax is allowed as in source files. Be careful with string quoting, the shell might eat some of the characters.

```
64tass -D ii=2 a.asm
        lda #ii ;result: $a9, $02
```

-w, --no-warn
: Suppress warnings. Disables warnings during compile.

```
64tass --no-warn a.asm
```

--no-caret-diag
: Suppress displaying of faulty source line and fault position after fault messages.

```
64tass --no-caret-diag a.asm
```

-q, --quiet
: Suppress messages. Disables header and summary messages.

```
64tass --quiet a.asm
```

-T, --tasm-compatible
: Enable TASM compatible operators and precedence

Switches the expression evaluator into compatibility mode. This enables ".", ":" and "!" operators and disables 64tass specific extensions, disables precedence handling and forces 16 bit unsigned evaluation (see "differences to original Turbo Assembler" below)

-I <path>
: Specify include search path

If an included source or binary file can't be found in the directory of the source file then this path is tried. More than one directories can be specified by repeating this option. If multiple matches exist the first one is used.

7.3 Target selection on command line

These options will select the default architecture. It can be overridden by using the `.cpu` directive in the source.

--m65xx

Standard 65xx (default). For writing compatible code, no extra codes. This is the default.

```
64tass --m65xx a.asm
        lda $14         ;regular instructions
```

-c, --m65c02

CMOS 65C02. Enables extra opcodes and addressing modes specific to this CPU.

```
64tass --m65c02 a.asm
        stz $d020       ;65c02 instruction
```

-c, --m65ce02

CSG 65CE02. Enables extra opcodes and addressing modes specific to this CPU.

```
64tass --m65ce02 a.asm
        inz
```

-i, --m6502

NMOS 65xx. Enables extra illegal opcodes. Useful for demo coding for C64, disk drive code, etc.

```
64tass --m6502 a.asm
        lax $14         ;illegal instruction
```

-t, --m65dtv02

65DTV02. Enables extra opcodes specific to DTV.

```
64tass --m65dtv02 a.asm
        sac #$00
```

-x, --m65816

W65C816. Enables extra opcodes, and full 16 MiB address space. Useful for SuperCPU projects.

```
64tass --m65816 a.asm
        lda $123456,x
```

-e, --m65el02

65EL02. Enables extra opcodes, useful RedPower CPU projects. Probably you'll need “--nostart” as well.

```
64tass --m65el02 a.asm
        lda 0,r
```

--mr65c02

R65C02. Enables extra opcodes and addressing modes specific to this CPU.

```
64tass --mr65c02 a.asm
        rmb 7,$20
```

--mw65c02

W65C02. Enables extra opcodes and addressing modes specific to this CPU.

```
64tass --mw65c02 a.asm
        wai
```

7.4 Source listing options

-l <file>, **--labels**=<file>

List labels into <file>. List global used labels to a file.

```
64tass -l labels.txt a.asm
*       = $1000
label   jmp label

result (labels.txt):
label           = $1000
```

--vice-labels

List labels in a VICE readable format.

```
64tass --vice-labels -l labels.txt a.asm
*       = $1000
label   jmp label

result (labels.txt):
al 1000 .label
```

-L <file>, **--list**=<file>

List into <file>. Dumps source code and compiled code into file. Useful for debugging, it's much easier to identify the code in memory within the source files.

```
64tass -L list.txt a.asm
*       = $1000
        ldx #0
loop    dex
        bne loop
        rts

result (list.txt):

;64tass Turbo Assembler Macro V1.5x listing file of "a.asm"
;done on Fri Dec  9 19:08:55 2005


.1000            a2 00          ldx #$00                ldx #0
.1002            ca             dex             loop    dex
.1003            d0 fd          bne $1002               bne loop
.1005            60             rts                     rts

;******  end of code
```

-m, **--no-monitor**

Don't put monitor code into listing. There won't be any monitor listing in the list file.

```
64tass --no-monitor -L list.txt a.asm

result (list.txt):

;64tass Turbo Assembler Macro V1.5x listing file of "a.asm"
;done on Fri Dec  9 19:11:43 2005
```

```
.1000           a2 00                           ldx #0
.1002           ca                      loop    dex
.1003           d0 fd                           bne loop
.1005           60                              rts

;******  end of code
```

-s, --no-source
: Don't put source code into listing. There won't be any source listing in the list file.

```
64tass --no-source -L list.txt a.asm

result (list.txt):

;64tass Turbo Assembler Macro V1.5x listing file of "a.asm"
;done on Fri Dec  9 19:13:25 2005


.1000           a2 00           ldx #$00
.1002           ca              dex
.1003           d0 fd           bne $1002
.1005           60              rts

;******  end of code
```

--tab-size=<number>
: By default the listing file is using a tab size of 8 to align the disassembly. This can be changed to other more favorable values like 4. Only spaces are used if 1 is selected. Please note that this has no effect on the source code on the right hand side.

7.5 Other options

-?, --help
: Give this help list. Prints help about command line options.

--usage
: Give a short usage message. Prints short help about command line options.

-V, --version
: Print program version

8 Messages

Faults and warnings encountered are sent to standard error for logging. To redirect them into a file append “2>filename.log” after the command. The format of messages is the following:

<filename>:<line>:<character>: <severity>: <message>

- filename: The name and path of source file where the error happened.
- line: Line number of file, starts from 1.
- character: Character in line, starts from 1. Tabs are not expanded.
- severity: Note, warning, error or fatal.
- message: The fault message itself.

The faulty line may be displayed after the message with a caret pointing to the error location.

```
a.asm:3:21: error: not defined 'label'
                 lda label
                     ^
a.asm:3:21: note: searched in the global scope
```

Lines containing macros are expanded whenever possible, but due to internal limitations referenced lines in relation to the actual fault will display without them.

8.1 Warnings

`directive ignored`
: an assembler directive was ignored for compatibility reasons.

`label not on left side`
: check if an instruction name was not mistyped and if the currect CPU has it, or remove white space before label

`long branch used`
: branch too long, so long branch was used (bxx ∗+5 jmp)

`possible jmp ($xxff) bug`
: yet another 65xx feature...

`processor program counter overflow`
: pc address was set back to the start of actual 64 KiB program bank

`top of memory exceeded`
: compile continues at the bottom ($0000)

8.2 Errors

`? expected`
: something is missing

`address not in processor address space`
: value larger than current CPU address space

`address out of section`
: moving the address around is fine, but do not place it before the section

`at least one byte is needed`
: the expression didn't yield any bytes

`branch crosses page`
: page crossing detected

`branch too far by ? bytes`
: can't branch that far

`can't calculate stable value`
: somehow it's impossible to calculate this expression

`can't calculate this`
: could not get any value, is this a circular reference?

`can't convert to a ? bit signed/unsigned integer`
: value out of range

`can't encode character $xx`
: can't translate character, not part of current encoding

`can't get absolute value of type '?'`
: value has no absolute value

`can't get boolean value of type '?'`
: conversion error

`can't get integer value of type '?'`

conversion error

`can't get length of type '?'`
: value has no length

`can't get sign of type '?'`
: value does not have a sign

`can't get size of type '?'`
: value has no size

`conflict`
: at least one feature is provided, which shouldn't be there

`constant too large`
: floating point overflow and other value out of range conditions

`division by zero`
: can't calculate this

`double defined escape`
: escape sequence already defined in another .edef

`double defined range`
: part of a character range was already defined by another .cdef

`duplicate definition`
: symbol defined more than once

`empty range not allowed`
: invalid range

`empty string not allowed`
: at least one character is required

`expected exactly/at least/at most ? arguments, got ?`
: wrong number of function arguments

`expression syntax`
: syntax error

`extra characters on line`
: there's some garbage on the end of line

`floating point overflow`
: infinity reached during a calculation

`general syntax`
: can't do anything with this

`index out of range`
: not enough elements in list

`instruction can't cross banks`
: this instruction is only limited to the current bank

`invalid operands to ? '?' and '?'`
: can't do this calculation with these values

`key error`
: not in dictionary

`label required`
: a label is mandatory for this directive

`last byte must not be gap`
: .shift or .shiftl needs a normal byte at the end

`logarithm of non-positive number`
: only positive numbers have a logarithm

`missing argument`
: not enough arguments supplied to function

`most significiant bit must be clear in byte`
: for .shift and .shiftl only 7 bit "bytes" are valid

`negative number raised on fractional power`
: can't calculate this

`no ? addressing mode for opcode`
: this addressing mode is not valid for this opcode

`not a bank 0 address`
: value must be a bank zero address

`not a data bank address`
: value must be a data bank address

`not a direct page address`
: value must be a direct page address

`not a key and value pair`
: dictionaries are built from key and value pairs separated by a colon

`not a one character string`
: only a single character string is allowed

`not allowed here: ?`
: do not use this directive here

`not defined '?'`
: can't find this label

`not hashable`
: can't be used as a key in a dictionary

`not in range -1.0 to 1.0`
: the function is only valid in the -1.0 to 1.0 range

`not iterable`
: value is not a list or other iterable object

`operands could not be broadcast together with shapes ? and ?`
: list length must match or must have a single element only

`page error at $xxxx`
: page crossing detected

`ptext too long by ? bytes`
: .ptext is limited to 255 bytes maximum

`requirements not met`
: Not all features are provided, at least one is missing

`reserved symbol name '?'`
: do not use this symbol name

`square root of negative number`
: can't calculate the square root of a negative number

`too early to reference`
: processing still ongoing, can't access this yet

`unknown processor '?'`
: unknown cpu name

`wrong type <?>`
: wrong object type used

`zero value not allowed`
: do not use zero, also not with .null

8.3 Fatal errors

`can't open file`
: cannot open file

`can't write label file`
: cannot write the label file

`can't write listing file`
: cannot write the list file

`can't write object file`
: cannot write the result

`error reading file`
: error while reading

`file recursion`
: wrong use of .include

`macro recursion too deep`
: wrong use of nested macros

`function recursion too deep`
: wrong use of nested functions

`unknown option '?'`
: option not known

`out of memory`
: won't happen ;)

`too many passes`
: with a carefully crafted source file it's possible to create unresolvable situations. Fix your code.

9 Credits

Original written for DOS by Marek Matula of Taboo, then ported to ANSI C by BigFoot/Breeze, and finally added 65816 support, DTV, illegal opcodes, optimizations, multi pass compile and a lot of features by Soci/Singular. Improved TASS compatibility, PETSCII codes by Groepaz.

Additional code: my_getopt command-line argument parser by Benjamin Sittler, avl tree code by Franck Bui-Huu, ternary tree code by Daniel Berlin, snprintf Alain Magloire, Amiga OS4 support files by Janne Peräaho.

Pierre Zero helped to uncover a lot of faults by fuzzing.

Main developer and maintainer: soci at c64.rulez.org

10 Default translation and escape sequences

10.1 Raw 8-bit source

By default raw 8-bit encoding is used and nothing is translated or escaped. This mode is for compiling sources which are already PETSCII.

10.1.1 The “none” encoding for raw 8-bit

Does no translation at all, no translation table, no escape sequences.

10.1.2 The “screen” encoding for raw 8-bit

The following translation table applies, no escape sequences.

Input	Byte	Input	Byte
00–1F	80–9F	20–3F	20–3F
40–5F	00–1F	60–7F	40–5F
80–9F	80–9F	A0–BF	60–7F
C0–FE	40–7E	FF	5E

Table 31: Built-in PETSCII to PETSCII screen code translation table

10.2 Unicode and ASCII source

Unicode encoding is used when the “-a” option is given on the command line.

10.2.1 The “none” encoding for Unicode

This is a Unicode to PETSCII mapping, including escape sequences for control codes.

Glyph	Unicode	Byte	Glyph	Unicode	Byte
-@	U+0020-U+0040	20-40	A-Z	U+0041-U+005A	C1-DA
[	U+005B	5B	]	U+005D	5D
a-z	U+0061-U+007A	41-5A	£	U+00A3	5C
π	U+03C0	FF	←	U+2190	5F
↑	U+2191	5E	─	U+2500	C0
│	U+2502	DD	┌	U+250C	B0
┐	U+2510	AE	└	U+2514	AD
┘	U+2518	BD	├	U+251C	AB
┤	U+2524	B3	┬	U+252C	B2
┴	U+2534	B1	┼	U+253C	DB
╭	U+256D	D5	╮	U+256E	C9
╯	U+256F	CB	╰	U+2570	CA
╱	U+2571	CE	╲	U+2572	CD
╳	U+2573	D6	▁	U+2581	A4
▂	U+2582	AF	▃	U+2583	B9
▄	U+2584	A2	▌	U+258C	A1
▍	U+258D	B5	▎	U+258E	B4
▏	U+258F	A5	▒	U+2592	A6
▔	U+2594	A3	▕	U+2595	A7
▖	U+2596	BB	▗	U+2597	AC
▘	U+2598	BE	▚	U+259A	BF
▝	U+259D	BC	○	U+25CB	D7
●	U+25CF	D1	◤	U+25E4	A9
◥	U+25E5	DF	♠	U+2660	C1
♣	U+2663	D8	♥	U+2665	D3
♦	U+2666	DA	✓	U+2713	BA

Table 32: Built-in Unicode to PETSCII translation table

Escape	Byte	Escape	Byte	Escape	Byte
{bell}	07	{black}	90	{blk}	90
{blue}	1F	{blu}	1F	{brn}	95
{brown}	95	{cbm-*}	DF	{cbm-+}	A6
{cbm--}	DC	{cbm-0}	30	{cbm-1}	81
{cbm-2}	95	{cbm-3}	96	{cbm-4}	97
{cbm-5}	98	{cbm-6}	99	{cbm-7}	9A
{cbm-8}	9B	{cbm-9}	29	{cbm-@}	A4
{cbm-^}	DE	{cbm-a}	B0	{cbm-b}	BF
{cbm-c}	BC	{cbm-d}	AC	{cbm-e}	B1
{cbm-f}	BB	{cbm-g}	A5	{cbm-h}	B4

Table 33: Built-in PETSCII escape sequences

Escape	Byte	Escape	Byte	Escape	Byte
{cbm-i}	A2	{cbm-j}	B5	{cbm-k}	A1
{cbm-l}	B6	{cbm-m}	A7	{cbm-n}	AA
{cbm-o}	B9	{cbm-pound}	A8	{cbm-p}	AF
{cbm-q}	AB	{cbm-r}	B2	{cbm-s}	AE
{cbm-t}	A3	{cbm-up arrow}	DE	{cbm-u}	B8
{cbm-v}	BE	{cbm-w}	B3	{cbm-x}	BD
{cbm-y}	B7	{cbm-z}	AD	{clear}	93
{clr}	93	{control-0}	92	{control-1}	90
{control-2}	05	{control-3}	1C	{control-4}	9F
{control-5}	9C	{control-6}	1E	{control-7}	1F
{control-8}	9E	{control-9}	12	{control-:}	1B
{control-;}	1D	{control-=}	1F	{control-@}	00
{control-a}	01	{control-b}	02	{control-c}	03
{control-d}	04	{control-e}	05	{control-f}	06
{control-g}	07	{control-h}	08	{control-i}	09
{control-j}	0A	{control-k}	0B	{control-left arrow}	06
{control-l}	0C	{control-m}	0D	{control-n}	0E
{control-o}	0F	{control-pound}	1C	{control-p}	10
{control-q}	11	{control-r}	12	{control-s}	13
{control-t}	14	{control-up arrow}	1E	{control-u}	15
{control-v}	16	{control-w}	17	{control-x}	18
{control-y}	19	{control-z}	1A	{cr}	0D
{cyan}	9F	{cyn}	9F	{delete}	14
{del}	14	{dish}	08	{down}	11
{ensh}	09	{esc}	1B	{f10}	82
{f11}	84	{f12}	8F	{f1}	85
{f2}	89	{f3}	86	{f4}	8A
{f5}	87	{f6}	8B	{f7}	88
{f8}	8C	{f9}	80	{gray1}	97
{gray2}	98	{gray3}	9B	{green}	1E
{grey1}	97	{grey2}	98	{grey3}	9B
{grn}	1E	{gry1}	97	{gry2}	98
{gry3}	9B	{help}	84	{home}	13
{insert}	94	{inst}	94	{lblu}	9A
{left arrow}	5F	{left}	9D	{lf}	0A
{lgrn}	99	{lower case}	0E	{lred}	96
{lt blue}	9A	{lt green}	99	{lt red}	96
{orange}	81	{orng}	81	{pi}	FF
{pound}	5C	{purple}	9C	{pur}	9C
{red}	1C	{return}	0D	{reverse off}	92
{reverse on}	12	{rght}	1D	{right}	1D
{run}	83	{rvof}	92	{rvon}	12
{rvs off}	92	{rvs on}	12	{shift return}	8D
{shift-*}	C0	{shift-+}	DB	{shift-,}	3C
{shift--}	DD	{shift-.}	3E	{shift-/}	3F
{shift-0}	30	{shift-1}	21	{shift-2}	22
{shift-3}	23	{shift-4}	24	{shift-5}	25
{shift-6}	26	{shift-7}	27	{shift-8}	28
{shift-9}	29	{shift-:}	5B	{shift-;}	5D
{shift-@}	BA	{shift-^}	DE	{shift-a}	C1
{shift-b}	C2	{shift-c}	C3	{shift-d}	C4
{shift-e}	C5	{shift-f}	C6	{shift-g}	C7
{shift-h}	C8	{shift-i}	C9	{shift-j}	CA
{shift-k}	CB	{shift-l}	CC	{shift-m}	CD

Escape	Byte	Escape	Byte	Escape	Byte
{shift-n}	CE	{shift-o}	CF	{shift-pound}	A9
{shift-p}	D0	{shift-q}	D1	{shift-r}	D2
{shift-space}	A0	{shift-s}	D3	{shift-t}	D4
{shift-up arrow}	DE	{shift-u}	D5	{shift-v}	D6
{shift-w}	D7	{shift-x}	D8	{shift-y}	D9
{shift-z}	DA	{space}	20	{sret}	8D
{stop}	03	{swlc}	0E	{swuc}	8E
{tab}	09	{up arrow}	5E	{up/lo lock off}	09
{up/lo lock on}	08	{upper case}	8E	{up}	91
{white}	05	{wht}	05	{yellow}	9E
{yel}	9E				

10.2.2 The "screen" encoding for Unicode

This is a Unicode to PETSCII screen code mapping, including escape sequences for control code screen codes.

Glyph	Unicode	Translated	Glyph	Unicode	Translated
-?	U+0020-U+003F	20-3F	@	U+0040	00
A-Z	U+0041-U+005A	41-5A	[	U+005B	1B
]	U+005D	1D	a-z	U+0061-U+007A	01-1A
£	U+00A3	1C	π	U+03C0	5E
←	U+2190	1F	↑	U+2191	1E
─	U+2500	40	│	U+2502	5D
┌	U+250C	70	┐	U+2510	6E
└	U+2514	6D	┘	U+2518	7D
├	U+251C	6B	┤	U+2524	73
┬	U+252C	72	┴	U+2534	71
┼	U+253C	5B	╭	U+256D	55
╮	U+256E	49	╯	U+256F	4B
╰	U+2570	4A	╱	U+2571	4E
╲	U+2572	4D	╳	U+2573	56
▁	U+2581	64	▂	U+2582	6F
▃	U+2583	79	▄	U+2584	62
▌	U+258C	61	▍	U+258D	75
▎	U+258E	74	▏	U+258F	65
▒	U+2592	66	▔	U+2594	63
▕	U+2595	67	▖	U+2596	7B
▗	U+2597	6C	▘	U+2598	7E
▚	U+259A	7F	▝	U+259D	7C
○	U+25CB	57	●	U+25CF	51
◤	U+25E4	69	◥	U+25E5	5F
♠	U+2660	41	♣	U+2663	58
♥	U+2665	53	♦	U+2666	5A
✓	U+2713	7A			

Table 34: Built-in Unicode to PETSCII screen code translation table

Escape	Byte	Escape	Byte	Escape	Byte
{cbm-*}	5F	{cbm-+}	66	{cbm--}	5C
{cbm-0}	30	{cbm-9}	29	{cbm-@}	64
{cbm-^}	5E	{cbm-a}	70	{cbm-b}	7F
{cbm-c}	7C	{cbm-d}	6C	{cbm-e}	71
{cbm-f}	7B	{cbm-g}	65	{cbm-h}	74
{cbm-i}	62	{cbm-j}	75	{cbm-k}	61
{cbm-l}	76	{cbm-m}	67	{cbm-n}	6A

Table 35: Built-in PETSCII screen code escape sequences

Escape	Byte	Escape	Byte	Escape	Byte
{cbm-o}	79	{cbm-pound}	68	{cbm-p}	6F
{cbm-q}	6B	{cbm-r}	72	{cbm-s}	6E
{cbm-t}	63	{cbm-up arrow}	5E	{cbm-u}	78
{cbm-v}	7E	{cbm-w}	73	{cbm-x}	7D
{cbm-y}	77	{cbm-z}	6D	{left arrow}	1F
{pi}	5E	{pound}	1C	{shift-*}	40
{shift-+}	5B	{shift-,}	3C	{shift--}	5D
{shift-.}	3E	{shift-/}	3F	{shift-0}	30
{shift-1}	21	{shift-2}	22	{shift-3}	23
{shift-4}	24	{shift-5}	25	{shift-6}	26
{shift-7}	27	{shift-8}	28	{shift-9}	29
{shift-:}	1B	{shift-;}	1D	{shift-@}	7A
{shift-^}	5E	{shift-a}	41	{shift-b}	42
{shift-c}	43	{shift-d}	44	{shift-e}	45
{shift-f}	46	{shift-g}	47	{shift-h}	48
{shift-i}	49	{shift-j}	4A	{shift-k}	4B
{shift-l}	4C	{shift-m}	4D	{shift-n}	4E
{shift-o}	4F	{shift-pound}	69	{shift-p}	50
{shift-q}	51	{shift-r}	52	{shift-space}	60
{shift-s}	53	{shift-t}	54	{shift-up arrow}	5E
{shift-u}	55	{shift-v}	56	{shift-w}	57
{shift-x}	58	{shift-y}	59	{shift-z}	5A
{space}	20	{up arrow}	1E		

11 Opcodes

11.1 Standard 6502 opcodes

ADC	61 65 69 6D 71 75 79 7D	**AND**	21 25 29 2D 31 35 39 3D
ASL	06 0A 0E 16 1E	**BCC**	90
BCS	B0	**BEQ**	F0
BIT	24 2C	**BMI**	30
BNE	D0	**BPL**	10
BRK	00	**BVC**	50
BVS	70	**CLC**	18
CLD	D8	**CLI**	58
CLV	B8	**CMP**	C1 C5 C9 CD D1 D5 D9 DD
CPX	E0 E4 EC	**CPY**	C0 C4 CC
DEC	C6 CE D6 DE	**DEX**	CA
DEY	88	**EOR**	41 45 49 4D 51 55 59 5D
INC	E6 EE F6 FE	**INX**	E8
INY	C8	**JMP**	4C 6C
JSR	20	**LDA**	A1 A5 A9 AD B1 B5 B9 BD
LDX	A2 A6 AE B6 BE	**LDY**	A0 A4 AC B4 BC
LSR	46 4A 4E 56 5E	**NOP**	EA
ORA	01 05 09 0D 11 15 19 1D	**PHA**	48
PHP	08	**PLA**	68
PLP	28	**ROL**	26 2A 2E 36 3E
ROR	66 6A 6E 76 7E	**RTI**	40
RTS	60	**SBC**	E1 E5 E9 ED F1 F5 F9 FD
SEC	38	**SED**	F8
SEI	78	**STA**	81 85 8D 91 95 99 9D
STX	86 8E 96	**STY**	84 8C 94

Table 36: The standard 6502 opcodes

TAX	AA	**TAY**	A8
TSX	BA	**TXA**	8A
TXS	9A	**TYA**	98
ASL	0A	**BGE**	B0
BLT	90	**GCC**	4C 90
GCS	4C B0	**GEQ**	4C F0
GGE	4C B0	**GLT**	4C 90
GMI	30 4C	**GNE**	4C D0
GPL	10 4C	**GVC**	4C 50
GVS	4C 70	**LSR**	4A
ROL	2A	**ROR**	6A
SHL	06 0A 0E 16 1E	**SHR**	46 4A 4E 56 5E

Table 37: Aliases, pseudo instructions

11.2 6502 illegal opcodes

This processor is a standard 6502 with the NMOS illegal opcodes.

ANC	0B	**ANE**	8B
ARR	6B	**ASR**	4B
DCP	C3 C7 CF D3 D7 DB DF	**ISB**	E3 E7 EF F3 F7 FB FF
JAM	02	**LAX**	A3 A7 AB AF B3 B7 BF
LDS	BB	**NOP**	04 0C 14 1C 80
RLA	23 27 2F 33 37 3B 3F	**RRA**	63 67 6F 73 77 7B 7F
SAX	83 87 8F 97	**SBX**	CB
SHA	93 9F	**SHS**	9B
SHX	9E	**SHY**	9C
SLO	03 07 0F 13 17 1B 1F	**SRE**	43 47 4F 53 57 5B 5F

Table 38: Additional opcodes

AHX	93 9F	**ALR**	4B
AXS	CB	**DCM**	C3 C7 CF D3 D7 DB DF
INS	E3 E7 EF F3 F7 FB FF	**ISC**	E3 E7 EF F3 F7 FB FF
LAE	BB	**LAS**	BB
LXA	AB	**TAS**	9B
XAA	8B		

Table 39: Additional aliases

11.3 65DTV02 opcodes

This processor is an enhanced version of standard 6502 with some illegal opcodes.

BRA	12	**SAC**	32
SIR	42		

Table 40: Additionally to 6502 illegal opcodes

GRA	12 4C		

Table 41: Additional pseudo instruction

ANC	0B	**JAM**	02
LDS	BB	**NOP**	04 0C 14 1C 80
SBX	CB	**SHA**	93 9F
SHS	9B	**SHX**	9E
SHY	9C		

Table 42: These illegal opcodes are not valid

AHX	93 9F	**AXS**	CB

Table 43: These aliases are not valid

LAE	BB	LAS	BB
TAS	9B		

11.4 Standard 65C02 opcodes

This processor is an enhanced version of standard 6502.

ADC	72	AND	32
BIT	34 3C 89	BRA	80
CMP	D2	DEC	3A
EOR	52	INC	1A
JMP	7C	LDA	B2
ORA	12	PHX	DA
PHY	5A	PLX	FA
PLY	7A	SBC	F2
STA	92	STZ	64 74 9C 9E
TRB	14 1C	TSB	04 0C

Table 44: Additional opcodes

CLR	64 74 9C 9E	DEA	3A
GRA	4C 80	INA	1A

Table 45: Additional aliases and pseudo instructions

11.5 R65C02 opcodes

This processor is an enhanced version of standard 65C02.

BBR	0F 1F 2F 3F 4F 5F 6F 7F	BBS	8F 9F AF BF CF DF EF FF
RMB	07 17 27 37 47 57 67 77	SMB	87 97 A7 B7 C7 D7 E7 F7

Table 46: Additional opcodes

11.6 W65C02 opcodes

This processor is an enhanced version of R65C02.

STP	DB	WAI	CB

Table 47: Additional opcodes

HLT	DB		

Table 48: Additional aliases

11.7 W65816 opcodes

This processor is an enhanced version of W65C02.

ADC	63 67 6F 73 77 7F	AND	23 27 2F 33 37 3F
BRL	82	CMP	C3 C7 CF D3 D7 DF
COP	02	EOR	43 47 4F 53 57 5F
JMP	5C DC	JSL	22
JSR	FC	LDA	A3 A7 AF B3 B7 BF
MVN	54	MVP	44
ORA	03 07 0F 13 17 1F	PEA	F4
PEI	D4	PER	62
PHB	8B	PHD	0B
PHK	4B	PLB	AB
PLD	2B	REP	C2
RTL	6B	SBC	E3 E7 EF F3 F7 FF

Table 49: Additional opcodes

SEP	E2	STA	83 87 8F 93 97 9F
TCD	5B	TCS	1B
TDC	7B	TSC	3B
TXY	9B	TYX	BB
XBA	EB	XCE	FB

CSP	02	CLP	C2
JML	5C DC	SWA	EB
TAD	5B	TAS	1B
TDA	7B	TSA	3B

Table 50: Additional aliases

11.8 65EL02 opcodes

This processor is an enhanced version of standard 65C02.

ADC	63 67 73 77	AND	23 27 33 37
CMP	C3 C7 D3 D7	DIV	4F 5F 6F 7F
ENT	22	EOR	43 47 53 57
JSR	FC	LDA	A3 A7 B3 B7
MMU	EF	MUL	0F 1F 2F 3F
NXA	42	NXT	02
ORA	03 07 13 17	PEA	F4
PEI	D4	PER	62
PHD	DF	PLD	CF
REA	44	REI	54
REP	C2	RER	82
RHA	4B	RHI	0B
RHX	1B	RHY	5B
RLA	6B	RLI	2B
RLX	3B	RLY	7B
SBC	E3 E7 F3 F7	SEA	9F
SEP	E2	STA	83 87 93 97
STP	DB	SWA	EB
TAD	BF	TDA	AF
TIX	DC	TRX	AB
TXI	5C	TXR	8B
TXY	9B	TYX	BB
WAI	CB	XBA	EB
XCE	FB	ZEA	8F

Table 51: Additional opcodes

CLP	C2	HLT	DB

Table 52: Additional aliases

11.9 65CE02 opcodes

This processor is an enhanced version of R65C02.

ASR	43 44 54	ASW	CB
BCC	93	BCS	B3
BEQ	F3	BMI	33
BNE	D3	BPL	13
BRA	83	BSR	63
BVC	53	BVS	73
CLE	02	CPZ	C2 D4 DC

Table 53: Additional opcodes

DEW	C3	**DEZ**	3B
INW	E3	**INZ**	1B
JSR	22 23	**LDA**	E2
LDZ	A3 AB BB	**NEG**	42
PHW	F4 FC	**PHZ**	DB
PLZ	FB	**ROW**	EB
RTS	62	**SEE**	03
STA	82	**STX**	9B
STY	8B	**TAB**	5B
TAZ	4B	**TBA**	7B
TSY	0B	**TYS**	2B
TZA	6B		
ASR	43	**BGE**	B3
BLT	93	**NEG**	42
RTN	62		

Table 54: Additional aliases

CLR	64 74 9C 9E		

Table 55: This alias is not valid

12 Appendix

12.1 Assembler directives

.addr .al .align .as .assert .autsiz .bend .binary .binclude .block .break .byte .case .cdef .cerror .char .check .comment .continue .cpu .cwarn .databank .default .dint .dpage .dsection .dstruct .dunion .dword .edef .else .elsif .enc .end .endc .endf .endif .endm .endp .ends .endswitch .endu .endweak .eor .error .fi .fill .for .function .goto .here .hidemac .if .ifeq .ifmi .ifne .ifpl .include .int .lbl .lint .logical .long .macro .mansiz .next .null .offs .option .page .pend .proc .proff .pron .ptext .rept .rta .section .seed .segment .send .shift .shiftl .showmac .struct .switch .text .union .var .warn .weak .word .xl .xs

12.2 Built-in functions

abs acos all any asin atan atan2 cbrt ceil cos cosh deg exp floor format frac hypot len log log10 pow rad random range repr round sign sin sinh size sqrt tan tanh trunc

12.3 Built-in types

address bits bool bytes code dict float gap int list str tuple type