max65 User Guide

`max65` is a command-line macro cross-assembler for the 65xx CPU family. It is useful for many systems but it specifically targets 8-bit Acorn computers like the Electron and BBC Micro.

This user guide explains how to use `max65` but is not a tutorial on 65xx assembly programming. There are many books and online resources about that.

Table of contents

Example program
Overview
Command-line options
Error and warning messages
Expressions
  Literals
  Symbols
    Predefined global symbols
    Symbol scopes
  Operators
  Built-in functions
Assembler directives
Macros
  Advanced macros
Comparing with BeebAsm
Features beyond BeebAsm
Supported instructions
  6502
  65C02
Quirks and tips
Download and install
  Windows
  Linux
  macOS
  Syntax highlighting
Changelog
Disclaimer
Contact

Example program

Let's dive right in with a short but real program to demonstrate some of the features of `max65`:

```
  \ define some constants
  RUN_ADDR=$0400              ; binary will execute at this address
  LOAD_ADDR=$2000             ; binary will load at this address
  OFFSET=LOAD_ADDR-RUN_ADDR   ; displacement between main program and relocator

  \ define zeropage variables
  org 0                       ; start assembling at address $0000 (default)
.sin skip 2                   ; reserve 2 bytes for zeropage variable 'sin'

  \ main program
  org RUN_ADDR                ; continue assembling at address 'RUN_ADDR'
.main                         ; named label 'main': start of main program
  lda #<S: sta sin+0          ; write low byte of 'S' to zeropage variable 'sin'
  lda #>S: sta sin+1          ; write high byte of 'S' to zeropage variable 'sin'
  ; defining 'S' *after* using it is perfectly legal (lazy expression evaluation)
  S=(1<<16)*sin(rad(deg(45))) ; set 'S': use complex expressions and built-in
functions
  rts                         ; return to caller
.end                          ; end of main program

  \ relocator stub that moves main program from 'LOAD_ADDR' to 'RUN_ADDR'
  org *, *+OFFSET             ; continue assembling at current PC (*)
                              ; also set logical PC (@) to where relocator will
execute
.entry                        ; program entry (execution starts here)
  ldx #end-main               ; number of bytes to copy (size of main program)
.                             ; anonymous (unnamed) label
  lda LOAD_ADDR-1,x           ; copy byte from 'LOAD_ADDR' area
  sta RUN_ADDR-1,x            ;           to 'RUN_ADDR' area
  dex                         ; point register X to next byte
  bne -                       ; branch to previously defined anonymous label
  jmp main                    ; finally, execute main program at 'RUN_ADDR'

  ; save the final binary named "CODE" from label 'main' to current PC (*)
  ; also save its accompanying .inf file
  ; set execution address to 'entry' (the $FF0000 signifies a host address)
  ; set load address to 'LOAD_ADDR' (again, the $FF0000 signifies a host address)
  save "CODE", main, *, $FF0000|entry, $FF0000|LOAD_ADDR
```

Overview

`max65` is heavily inspired by BeebAsm and aims to improve and extend it. If you are familiar with BeebAsm and/or BBC BASIC, many assembler directives and built-in functions are instantly recognisable. For a quick start, see Comparing with BeebAsm and Features beyond BeebAsm.

> I did not write `max65` because there is a shortage of 65xx assemblers -- far from it. I wrote `max65` because it seemed like a fun and challenging project. And I was right!

`max65` is a two-pass assembler. In pass 1 source files are tokenised and tokens are parsed to internal commands. In pass 2 these internal commands are translated to machine code and data. You can then save any part(s) of the 64Kb memory map to your computer as raw binary file(s).

A source file is a plain text file consisting of 65xx instructions, assembler directives, label definitions and symbol assignments. Each element may be put on a separate line. A single line may also contain multiple elements, separated by colons (`:`). A source file also usually contains whitespace and comments.

This example shows all elements in a source file:

```
  org &e00       ; assembler directive
.start           ; label definition
  NUM=%1000_0000 ; symbol assignment
  lda #NUM>>2    ; 65xx instruction
```

`max65` uses lazy expression evaluation. The main benefit for the user is that any symbol may be used (forward referenced) before being defined. The only exception is that macros must be defined before use. In pass 2 the final evaluation of expressions is done and all symbols must be resolved then of course. Example of forward referencing:

```
  ; All symbols below are used before being defined
  lda data,x
  sta &5800+N*320
  rts
.data equb D1, D2, D3
  D1=D3-N: D2=D1*2: D3=42: N=10
```

In some cases `max65` needs to make an educated guess in pass 1 about forward references, especially when the choice between zeropage and absolute addressing modes needs to be made. It is therefore recommended (but not required) to define zeropage labels as early as possible in your program.

Command-line options

Here is a brief summary of how to invoke `max65`:

```
max65 [-D <sym>=<expr>] [-h] [-l <listfile>] [-O] [-v] <infile>
```

Option	Description
`<infile>`	Input file (plain text source)
`-D <sym>=<expr>`	Define symbol with the given expression
`-h`	Show a help message and exit
`-l <listfile>`	Create a listing file
`-O`	Show potential code optimisations
`-v`	Enable verbose output

Example command-line: `max65 -D DBG=1 -v -D MSG=\"hello\" main.6502 -l listing.txt -O`.

Error and warning messages

When all goes well `max65` happily assembles the source file and exits with exit code 0 (success). However, when `max65` fails to assemble the source file, an error message is written to the standard error file (stderr) and the assembler exits with exit code 1 (failure). An error message shows the source filename, line number and cause of the error. Example of a typical error message: `*** error in pass 2: file main.6502, line 10: operator '/' doesn't work on strings`.

In a few cases `max65` writes a warning message to stderr, but continues assembling your program. Example of a typical warning message: `*** warning in pass 2: file main.6502, line 6: instruction 'lda' can be assembled 1 byte shorter in zeropage addressing mode`. Set the warning level with the `warn` directive.

Expressions

`max65` can handle arbitrarily complex expressions in directives, symbol assignments or 65xx instructions. Expressions consist of literals, symbols, operators and built-in functions. Each expression must eventually evaluate to an integer, float or string in pass 2.

Literals

You can use integer, float and string literals in expressions.

- **Integers** are positive or negative whole numbers, of just about any size. In practice you will use 8-bit, 16-bit or 32-bit integers though. You may use decimal notation (e.g. `123`, `-64`), hexadecimal notation (e.g. `$FFEE`, `-&ac`), binary notation (e.g. `%1101`, `-%11110`) or char notation (e.g. `'a'`, `-'C'`).
  Use an underscore (_) or a single quote (') to group digits, e.g. `%110_001`, `&FE'03`, `12'34_56`.
  Some chars need to be escaped with a backslash (\): `'\\'`, `'\''`. Empty chars (`''`) are not allowed.

- **Floats** are positive or negative fractional numbers, of just about any size. Only decimal notation is supported with at least 1 digit before and after the decimal point, e.g. `3.1415`, `-5.0`, but not `.001`. Scientific notation (e.g. `1.23E-6`) is not supported.
  In many cases only the integer part of a float will be used automatically. For instance, `ldx #3.14` will be assembled as `ldx #3`.

- **Strings** are sequences of 0 or more characters, enclosed in double quotation marks. Examples: `""`, `"Hello!"`, `"This is a backslash: \\"`. Unlike an empty char (`''`), an empty string (`""`) is perfectly allowed. Some characters in a string need to be escaped with a backslash (\): `'\\'`, `'\"'`.

Symbols

Symbols are used to name things in a source file. The name of a symbol is case sensitive and consists of any mix of digits, letters and underscores. It must not start with a digit and it optionally ends with a `%`. Examples of valid symbol names are: `_`, `Lives%`, `data_Table4`. A symbol has a specific scope in which it is valid and it (eventually) always refers to an integer, float or string. There are some predefined global symbols.

> Once defined, the value of a symbol cannot be changed by the programmer.

A symbol is defined in 1 of 5 ways:

1. **Label definition**. A label is defined by a period (`.`) directly followed by a symbol name, e.g. `.sine256`, `.current_level%`. The symbol is added to the internal nametable at the current scope level, unless it already exists there. Its value is set to the current logical program counter (`@`) which is a 16-bit integer. You can also define anonymous (unnamed) labels by simply using a period without name:

```
.       ; anonymous label #1
tya
beq +   ; jump to anonymous label #2
txa
beq ++  ; jump to anonymous label #3
.       ; anonymous label #2
iny
bpl --  ; jump to anonymous label #1
.       ; anonymous label #3
dex
bmi --- ; jump to anonymous label #1
```

2. **Symbol assignment**. An assignment consists of a symbol name, followed by an equal sign (`=`), and an expression. Examples: `N%=3`, `start_addr=&5800+N%*&140`. The expression stored with the symbol in the nametable can contain forward references but must eventually evaluate to an integer, float or string.

3. **Assembler directive `for`**. A `for...next` loop defines a local scope for every cycle of the loop with the `for`-loop variable (integer or float) as a local symbol. Example: `for n, 1, 10 ... next`. The body of the loop is assembled 10 times within the context of that local scope. The local symbol `n` increments from 1 to 10 during that time.

4. **Macro definition**. When you define a macro, e.g. `macro add num1, num2 ... endmacro`, a special global symbol based on the macro name is created in the internal nametable (`@add` in this example), unless a macro with that name already exists. Macro names therefore never clash with other symbols and also don't evaluate to any value by itself.

5. **Macro call**. When you invoke a previously defined macro, e.g. `add 12, 34`, a local scope is created with local symbols that are named after the parameters (if any) in the macro definition: `num1` and `num2` in this example. Their values are set to the macro call arguments, `12` and `34` in this example. The macro body is then assembled within the context of that local scope.

Predefined global symbols

`max65` has some predefined global symbols:

Symbol	Type	Value
`PI`	Float	3.141592653589793
`FALSE`	Integer	0

Symbol	Type	Value
TRUE	Integer	-1
* (or P%)	Integer	Current PC
@	Integer	Current logical PC
VERSION	Integer	Version of max65, e.g. $010A is version 1.10

Symbol scopes

You can create virtually unlimited nested local scopes for your symbols using curly braces, e.g.:

```
N=-1                          ; global symbol 'N'
.lab                          ; global label 'lab'
{
  N=3                         ; local symbol 'N', overrides global symbol 'N'
  for n, 1, N: equb n: next   ; local symbol 'n' for each loop cycle
  { .lab print N: N=6 }       ; local label 'lab' and another local symbol 'N'
                              ; overrides global label 'lab'
                              ; and overrides symbol 'N' in parent scopes
}
```

Defining symbols for a different scope within the current scope is also possible by using a special notation, e.g.:

```
{
  N*=32         ; define global symbol 'N'
  {
    .^lab       ; define local label 'lab' that is visible in current and parent
scope
    .*lab2      ; define global label 'lab2'
    { N^=-99 }  ; define local symbol 'N' that is visible in current and parent
scope
    print N     ; -99
  }
}
print N         ; 32
```

Anonymous labels (.) can be defined anywhere, but you can only branch to those in the current scope.

Operators

Operators in order of increasing precedence:

Operator	Associativity	Description

Operator	Associativity	Description
`<`, `>`	Right	Equal to `lo()` and `hi()` respectively, e.g. `lda #<&5800+n*320` is the same as `lda #lo(&5800+n*320)`
`or`	Left	Logical OR, e.g. `if addr>=&5800 or addr<&3000 ... endif`
`and`	Left	Logical AND, e.g. `if N==3 and DEBUG ... endif`
`|`	Left	Bitwise OR, e.g. `lda #1|2|4|&80`
`^`	Left	Bitwise XOR (EOR), e.g. `ldx #n^255`
`&`	Left	Bitwise AND, e.g. `lda #N&3`
`==`, `=`, `!=`, `<>`	Left	(In)equality, e.g. `assert N!=3 and DBG=2`
`<`, `<=`, `>`, `>=`	Left	Comparison, e.g. `if addr>=&5800 or addr<&3000 ... endif`
`+`, `-`	Left	Addition and subtraction, e.g. `equw 4*WIDTH-(N+8)`, `print "S="+lower$(S)`
`*`, `/`, `div`, `mod`	Left	Multiplication and division, e.g. `equb 8*(y div 8)`, `S=3*"ABC"`
`<<`, `>>`	Left	Bit and string shift, e.g. `lda #1<<5-1`, `equs "Hello world">>6`
`not`, `~`, `-`	Right	Logical NOT, bitwise NOT and unary minus, e.g. `if not defined("N") ... endif`, `lda #Q&~3`

Built-in functions

Built-in functions and expressions in parentheses (or, alternatively, in square brackets) have the highest priority:

Function	Description
`lo()`	Least significant byte (lower 8 bits), e.g. `adc #lo(SCRN)`
`hi()`	Most significant byte (upper 8 bits). Technically bits 15..8. E.g. `equb hi(SCRN+3*&140)`
`sin()`	Sine of an angle in radians, e.g. `equw 512*sin(t)`
`cos()`	Cosine of an angle in radians, e.g. `N=1024*cos(PI/4)`
`tan()`	Tangent of an angle in radians, e.g. `equd 32*tan(2*t)`
`asn()`	Arc sine (only defined for domain [-1, 1]), e.g. `n=asn(-0.5)`
`acs()`	Arc cosine (only defined for domain [-1, 1]), e.g. `n=acs(0.3*x)`
`atn()`	Arc tangent, e.g. `T=atn(100)`
`deg()`	Convert radians to degrees, e.g. `d=deg(PI/8)`

Function	Description
`rad()`	Convert degrees to radians, e.g. `r=rad(180)`
`int()`	Truncate float to integer, e.g. `lda #int(3.14)`
`abs()`	Absolute value, e.g. `m=abs(sin(t))`
`sqr()`	Square root (only defined for values >=0), e.g. `rt=sqr(36)`
`sgn()`	Sign of its argument: -1 for negative values, 0 for zero, 1 for positive values. E.g. `p=x+16*sgn(n)`
`log()`	Logarithm (base 10, only defined for values >0), e.g. `equb log(1000)`
`ln()`	Natural logarithm (base *e*, only defined for values >0), e.g. `equb ln(x/3)`
`exp()`	*e* raised to the power of the argument, e.g. `e=exp(1)`
`rnd()`	Random number. Only defined for integer values >0. `rnd(1)` returns a float in the range [0, 1]. `rnd(n)` with integer `n`>1 returns an integer in the range [1, `n`]. You can seed the random number generator with the `randomize` directive
`val()`	String to number. Checks if a string starts with a valid integer or float and returns that (or 0 if no number was found). E.g. `P=val("-3.14PI!")`
`len()`	Length of a string, e.g. `print len("Hello!")`
`asc()`	ASCII value of first character of a string (or -1 for an empty string), e.g. `val=asc("Hello!")`
`str$()`	Convert number to a string, e.g. `S=str$(123.4)>>1`
`str$~()`	Convert number to a string with the number in hexadecimal format, e.g. `print "$", str$~(128)`. Note: negative numbers are not printed in two's complement, so for example `str$~(-140)` translates to the string `"-8C"`, not `"FFFFFF74"` (which assumes 32-bit)
`chr$()`	Convert an ASCII value to a string of length 1 containing that ASCII character. Only valid for domain [32, 126]. E.g. `S=chr$(122)`
`lower$()`	Convert a string to lowercase, e.g. `S=lower$("ABC123")`
`upper$()`	Convert a string to uppercase, e.g. `S=upper$("abc123")`
`time$()`	Date and time of assembly (string). The argument is a string that determines the date/time format as specified by the Python (or C) function `strftime()`. `time$("")` returns date/time formatted as `"%a,%d %b %Y.%H:%M:%S"`, e.g. `"Mon,16 Jan 2023.17:46:03"`
`defined()`	Check if symbol is defined (returns `TRUE`) or not (returns `FALSE`), e.g. `if not defined("SCRN") ... endif` or `if defined("@my_macro") ... endif`

Assembler directives

Directives (or pseudo-ops) control the assembly process. A directive takes zero or more arguments. For instance, `skip 64` tells `max65` to skip 64 bytes. An argument is an expression that must evaluate to an integer, float or string. Floats are truncated when an integer is expected.

Sometimes the value of an argument must be known in pass 1 to let the assembler make the right decision. This means expressions must not contain unresolved symbols (forward references). For instance, in an `include` directive, `max65` must immediately know the name of the source file to include.

Other directives, like `print`, only evaluate their arguments in pass 2. In this case expressions that still have unresolved symbols (forward references) after pass 1 are allowed.

Directives that evaluate their arguments in pass 1:

Directive	Description
align *expr*	Increment PC to next multiple of *expr*, e.g. `align 256` aligns PC to the next memory page
cpu *expr*	Select allowed instruction set (default: 6502). If *expr* evaluates to 0, the 6502 instruction set is selected. For value 1, the 65C02 instruction set is selected
elif *expr*	Mark the start of an `elif`-block in an `if...endif`. See `if`
else	Mark the start of an `else`-block in an `if...endif`. See `if`
endif	Mark the end of an `if...endif` block. See `if`
endmacro	Mark the end of a macro definition. See `macro`
equb *expr_1 [, expr_2, ..., expr_n]*	Insert one or more bytes and/or strings, e.g. `equb "Hello!", 13, 10, 0`. For numeric arguments forward references are allowed
equs	Equivalent to `equb`
for *sym, expr_1, expr_2 [, expr_3] ...* **next**	Assemble a block of code/data one or more times. The loop counter *sym* is a local symbol which changes from *expr_1* to *expr_2* (inclusive) in steps of *expr_3* (-1 or 1 if unspecified). Examples: `for n, 0, 9: equb n: next`. Or: `for f, 3.5, -1.5, -0.75: print f: next`
if *expr_1 ...* *[***elif** *expr_2 ...] ... [***elif** *expr_n ...]* *[***else** *...]* **endif**	Conditional assembly. Assemble the code/data block for which the corresponding `if`-condition or the first `elif`-condition (in order of appearance) evaluates to a non-zero number (`TRUE`). When none exists, assemble the `else`-block (if any). Examples: `if n>3: print n: endif`. Or: `if n>3: print n: elif n<-3: print -n: else: print "Invalid": endif`
incbin *expr_1 [, expr_2 [, expr_3]]*	Insert binary file named *expr_1* and optionally specify start offset *expr_2* and length *expr_3*, e.g. `incbin "data/table.bin", $1000, 512`. *expr_1* is a string that contains a valid path to an existing file
include *expr*	Assemble and insert source file named *expr*, e.g. `include "src/spriteplot.asm"`. *expr* is a string that contains a valid path to an existing file

Directive	Description
macro *sym_1 [, sym_2, ..., sym_n] ...* **endmacro**	Define a macro named *sym_1* with optional parameters named *sym_2*, ..., *sym_n*. See also: Macros
next	Mark the end of a `for...next` loop. See `for`
org *expr_1 [, expr_2]*	Set PC (where code/data is assembled) to *expr_1*. Also set the logical PC (on which labels are based) to *expr_2*. If *expr_2* is not specified, the logical PC will be equal to PC. Examples: `org $1000`. Or: `org $1000, $2000`
skip *expr*	Increment PC by *expr* bytes, e.g. `skip 16`

Directives that evaluate their arguments in pass 2:

Directive	Description
assert *expr_1 [, expr_2, ..., expr_n]*	Evaluate one or more arguments and trigger an error for the first expression (in order of appearance) that is zero (`FALSE`). Example: `assert N<128, P%<$1000`
canvas *expr*	Set fill value (default: 0) for unused bytes to *expr*, e.g. `canvas &ff`. Used by `save` and `copyblock`
clear *expr_1, expr_2*	Clear a block of the memory map from *expr_1* to *expr_2* (exclusive), which allows you to assemble over it again. Does NOT clear any address guards set or any labels defined in that block. Example: `clear &1000, &2000`
copyblock *expr_1, expr_2, expr_3*	Copy a block of the memory map, ranging from *expr_1* to *expr_2* (exlusive), to destination *expr_3*. Example: `copyblock &1000, &2000, &2200`. Overwriting existing code/data is not allowed. Use `clear` first if necessary. Parts of the memory block with no code or data are filled with the fill value (default: 0) set by `canvas`
error *[expr_1, ..., expr_n]*	Trigger an error after printing *expr_1*, ..., *expr_n* to the standard error file (stderr). Example: `if N>=128: error "Expected N<128, but N=", N: endif`
equd *expr_1 [, expr_2, ..., expr_n]*	Insert one or more double words (32-bit integers), e.g. `equd $C0DE6502`

Directive	Description
equw *expr_1 [, expr_2, ..., expr_n]*	Insert one or more words (16-bit integers), e.g. `equw $C0DE`
export *expr [, sym_1, ..., sym_n]*	Export globals *sym_1*, ..., *sym_n* (named global labels, global constants and macros) to a plain text file named *expr*. Example: `export "inc/globals.txt", code_start, code_end, @my_macro, N_LIVES%`. Export ALL globals if no symbols are specified, e.g. `export "inc/globals.txt"`. To import these globals elsewhere, simply use the `include` directive
guard *[expr_1, ..., expr_n]*	Set multiple guards at addresses *expr_1*, ..., *epxr_n*. An address guard triggers an error when code/data is assembled at that address. When no arguments are given, clear all current guards. Example: `guard $5800, $8000`
print *[expr_1, ..., expr_n]*	Print zero or more expressions *expr_1*, ..., *expr_n* to the standard output file (stdout) and finish with a newline. When no arguments are given, just print a newline
randomize *expr*	Seed the random number generator with *expr*, e.g. `randomize 12345`. *expr* can be any integer, float or string
save *expr_1, expr_2, expr_3 [, expr_4 [, expr_5]]*	Save a code/data block from the 64Kb memory map to a raw binary file named *expr_1*. The block starts at *expr_2* and ends at *expr_3* (exclusive). The optional execution address *expr_4* and load address *expr_5* are used in the accompanying .inf file. When not specified, these are equal to *expr_2*. Example: `save "CODE", $1000, $2000, $f25, $e00`. Parts of the memory block with no code or data are filled with the fill value (default: 0) set by `canvas`
warn *[expr]*	Set warning level. If *expr* evaluates to 0, no warnings are shown. For value 1 (default), warnings are shown. `warn` without arguments restores the previous warning level

Macros

Macros are user defined code/data blocks and can be inserted anywhere in your program using a macro call. A macro is always global and must be defined before use. It takes zero or more parameters. Here is an example of a macro definition and a macro call:

```
\ macro definition
macro add_ptr ptr, val    ; 2 parameters
  lda ptr
  clc
  adc #lo(val)
  sta ptr
  lda ptr+1
```

```
  adc #hi(val)
  sta ptr+1
endmacro

\ macro call
add_ptr &70, &140          ; 2 arguments are passed to the macro
```

The macro call defines a local scope and binds the given arguments to the macro parameters. The macro call above therefore expands to:

```
{
  ptr=&70
  val=&140
  lda ptr
  clc
  adc #lo(val)
  sta ptr
  lda ptr+1
  adc #hi(val)
  sta ptr+1
}
```

A macro can call other macros and even itself recursively.

Advanced macros

A macro doesn't have to emit code or data and can be used as a sort of user defined function.

> Geek speak: this is done by exploiting macro call recursion, nested local scopes and defining symbols in parent scopes.

Here is an example of a user defined recursive Fibonacci function:

```
macro fib n
  if n<0: error "macro fib: negative number (", n, ") not allowed!"
  elif n<2: fib_result^=n
  else ; n>=2
    fib_result^=A+B
    { fib n-2: A^=fib_result }
    { fib n-1: B^=fib_result }
  endif
endmacro

for n, 0, 10
  fib n ; sets 'fib_result'
  print "fib(" , n, ")=", fib_result
next
```

One more example. A "raise to the power of" function taking 2 arguments:

```
macro pow x, y
  if y==0: pow_result^=1
  elif y<0
    pow_result^=res
    { pow x, y+1: res^=pow_result/x }
  else ; y>0
    pow_result^=res
    { pow x, y-1: res^=x*pow_result }
  endif
endmacro

for x, 5, 15, 5
  for y, -3, 3
    pow x, y ; sets 'pow_result'
    print "pow(", x, ", ", y, ")=", pow_result
  next
next
```

Comparing with BeebAsm

`max65` follows BeebAsm's syntax closely, but there are some differences and alternatives:

BeebAsm	**max65**
`LEFT$("abcde", 3)` (equals `"abc"`)	`"abcde">>2`
`RIGHT$("abcde", 3)` (equals `"cde"`)	`"abcde"<<2`
`MID$("abcde", 2, 3)` (equals `"bcd"`)	`"abcde"<<1>>1`
`STRING$(3, "ABC")` (equals `"ABCABCABC"`)	`"ABC"*3`
`N=?3` (define `N` if not defined yet)	`if not defined("N"): N=3: endif`
`COPYBLOCK`	Overwriting code/data at destination not allowed
`x^y` (raise to the power of)	`exp(y*ln(x))` for `x`>0
`EVAL()`	N/A
`TIME$`	`TIME$("")`
`AND` (logical)	`and`
`AND` (bitwise)	`&`
`OR` (logical)	`or`
`OR` (bitwise)	`|`
`EOR` (logical)	N/A
`EOR` (bitwise)	`^`

BeebAsm	max65
NOT (logical)	not
NOT (bitwise)	~
SKIPTO $2000	org $2000
CLEAR	Doesn't clear any address guards set
MAPCHAR	N/A
PRINT ~200 (equals "&C8")	print "&"+str$~(200)
ASM()	N/A
FILELINE$	N/A
CALLSTACK$	N/A
"AB""CD" (quote doubling)	"AB\"CD" (escape char)
PUTTEXT	N/A (no .ssd disk image I/O)
PUTFILE	N/A (no .ssd disk image I/O)
PUTBASIC	N/A (no .ssd disk image I/O)
RND(100) (random integer x, 0<=x<=99)	1<=x<=100 (like BBC BASIC)

Features beyond BeebAsm

max65 extends or improves BeebAsm in several ways:

Feature	Description
Undocumented 6502 instructions	alr, anc, ane, arr, dcp, dop, isc, jam, las, lax, nop, rla, rra, sax, sbc, sbx, sha, shx, shy, slo, sre, tas, top
Undocumented 65C02 instructions	dop, nop, top
N^=3	Define N in parent scope (and in current scope)
N*=3	Define N in global scope
&C0_DE, $6'502, 1'234, 3.14_15, %00_10'00	Digit grouping with _ and ' for all numbers, not just binary

Feature	Description
`N=A*A: A=2` (forward references in assignments)	Lazy expression evaluation allows forward references everywhere
`N=A*A: A=2*N`	Error on detection of circular references
`macro m: m: endmacro: m`	Error on detection of runaway recursion
in file a.asm: `include "a.asm"`	Error on detection of circular `include`s
`org $200, $500`	Optional second argument in `org` directive sets logical PC (`@`), e.g. assemble at `$200`, but labels are based on `$500`. Almost like `COPYBLOCK` in BeebAsm, or `O%` in BBC BASIC
`<<`, `>>` and `*` work on strings	`"abc"<<1` equals `"bc"`, `"abc">>2` equals `"a"`, `"A"*3` equals `"AAA"`
`{ ... include ... }`	Including source files works on any scope level (curly braces), and inside `for...next` loops as well
`randomize`	Seed the random number generator with any integer, float or string
`error`	Accepts zero or more arguments so it works similar to the `print` directive
`defined("N")`	`TRUE` if symbol `N` is defined, `FALSE` otherwise. Can also be used for macros, e.g. `if defined("@my_macro") ... endif`
`guard`	The `guard` directive sets one or more guards on the supplied memory addresses. When no arguments are given, all guards are cleared
zeropage vs absolute	`max65` issues a friendly warning (depending on the warning level set by `warn`) when an instruction could have used zeropage addressing mode (saving 1 byte)
forced absolute addressing	Place an exclamation mark (`!`) after a 65xx instruction to force absolute instead of zeropage addressing mode, e.g. `lda! 0` assembles to `ad 00 00` instead of `a5 00`
`.` (anonymous labels)	Use `.` to define unnamed labels. Relative branch instructions can jump backward or forward to them, e.g. `bne -`, or `bpl ++`
numbers	When an integer is expected, a float number is automatically truncated (not rounded) to an integer. Large integers and negative integers are allowed for 65xx instructions and directives like `equb`/`equw`/`equd`. E.g. `lda #-2` is equal to `lda #&fe`, `ldx #&123` is equal to `ldx #&23` (lower 8 bits), `equw $123456` is equal to `equw $3456` (lower 16 bits)
`canvas`	The `canvas` directive sets the fill value (default: 0) used for unused bytes. Used by `save` and `copyblock` to fill areas without code/data

Feature	Description
user defined functions	Sort of. See Advanced macros
`export`	Export (a selection of) globals (named global labels, global constants and macros) to a plain text file, that can be `include`d again elsewhere
`warn`	Set warning level. Default: 1 (show warnings)

Supported instructions

6502

`max65` supports all documented and undocumented 6502 instructions.

Documented 6502 instructions:
`adc`, `and`, `asl`, `bcc`, `bcs`, `beq`, `bit`, `bmi`, `bne`, `bpl`, `brk`, `bvc`, `bvs`, `clc`, `cld`, `cli`, `clv`, `cmp`, `cpx`, `cpy`, `dec`, `dex`, `dey`, `eor`, `inc`, `inx`, `iny`, `jmp`, `jsr`, `lda`, `ldx`, `ldy`, `lsr`, `nop`, `ora`, `pha`, `php`, `pla`, `plp`, `rol`, `ror`, `rti`, `rts`, `sbc`, `sec`, `sed`, `sei`, `sta`, `stx`, `sty`, `tax`, `tay`, `tsx`, `txa`, `txs`, `tya`.

Undocumented 6502 instructions:
`alr`, `anc`, `ane`, `arr`, `dcp`, `dop`, `isc`, `jam`, `las`, `lax`, `nop`, `rla`, `rra`, `sax`, `sbc`, `sbx`, `sha`, `shx`, `shy`, `slo`, `sre`, `tas`, `top`.

65C02

`max65` supports all documented and undocumented 65C02 instructions.

Documented 65C02 instructions:
All documented 6502 instructions and `bra`, `phx`, `phy`, `plx`, `ply`, `stz`, `trb`, `tsb`. But not `bbr`, `bbs`, `rmb`, `smb` (Rockwell, WDC) and `stp`, `wai` (WDC).

Undocumented 65C02 instructions:
`dop`, `nop`, `top`.

Quirks and tips

- It is best to use forward slashes (`/`) only in file paths, e.g. `include "src/prog.asm"`, `incbin "../data.bin"`. Using whitespace in file paths is discouraged.
- In an `if`-block or `elif`-block where the condition evaluates to zero (`FALSE`), chars and strings still need to be valid because of how the tokeniser works.
- Everything is case insensitive except for symbols which are case sensitive. For example, `guard`, `Guard` and `GUARD` all refer to the same directive. Similarly, `lda`, `LDA` and `LdA` all refer to the same instruction. But `sym1`, `Sym1` and `SYM1` are 3 different symbols.
- Use `canvas` and `copyblock` to fill a part of the memory map. Example: `canvas &55: copyblock &4000, &4300, &4000` (assuming no code/data was in this memory block yet).
- Assembling something like `if not defined("S"): S=123: endif` will always generate the warning "value for 'if' directive has changed between passes". You can safely ignore that or disable it (locally) with the `warn` directive, e.g. `warn 0: if not defined("S"): S=123: endif: warn`.

Download and install

The latest release of `max65` can always be found on GitHub.

64-bit binaries of `max65` are available for Windows and Linux (amd64). On macOS (and Linux as well) you can run `max65` by using Wine and the Windows binary.

Windows

Download the .zip file, extract it and optionally add the path to `max65.exe` to your system path. The assembler is now ready for use. You will also find this user guide in various formats in the `max65` folder.

The assembler is compiled and tested on 64-bit Windows 11. It is fully portable as long as you keep `max65.exe` together with the accompanying `python*.dll` and `python*.zip` files.

Linux

`max65` is available in the Snap Store. Use `snap install max65` to install it. The assembler is now ready for use. You will also find this user guide in various formats in the `/snap/max65/current` folder.

Alternatively, download the snap package from GitHub and use `snap install <filename> --dangerous` to install it.

The Windows binary is also known to work on Ubuntu 20.04.5 with Wine 5.0-3 and on Ubuntu 22.04.1 with Wine 6.0.3. I am confident that other combinations of Linux and Wine will work equally well.

macOS

The Windows binary has been tested and found working on macOS 13 (Ventura) with Wine 8.0. Again, I am confident that other combinations of macOS and Wine may work too.

For the record, I used `max65` on the following configuration: fresh install of macOS 13 (Ventura, x86_64), Xcode Command-line Tools 14.3, Homebrew 4.0.4 and Wine 8.0. This is how to install Wine and run `max65` (ignore Wine preloader warnings):

```
brew install --cask --no-quarantine wine-stable
WINEDEBUG=-all wine64 max65.exe
```

Syntax highlighting

You can use this extension for Visual Studio Code to enable syntax highlighting for `max65` compatible source files.

Changelog

Version	Date	Changes

Version	Date	Changes
0.17	Mar 9, 2023	Added `-O` option to show potential code optimisations `warn` directive sets warning level User guide: link to VSCode extension for `max65` syntax highlighting
0.16	Mar 5, 2023	Export (a selection of) globals with the `export` directive `clear` directive clears block of code/data in memory map `copyblock` directive copies block of code/data `skip` and `align` directives no longer fill the skipped bytes `filler` directive is renamed to `canvas` `%` is optionally allowed at the end of a symbol name User guide: how to run `max65` in macOS + Wine
0.15	Feb 28, 2023	Added 65C02 instruction set (not Rockwell/WDC) `cpu` directive selects 6502 (default) or 65C02 Fixed slow assembly when file has a large number of local scopes User guide: list all supported 6502 and 65C02 instructions
0.14	Feb 25, 2023	Fixed some undocumented 6502 instructions Fixed macro expansion `$.` prefix in .inf files Listing: can show labels +1 or +2 User guide: using macros as user defined functions
0.13	Feb 22, 2023	Define symbols on the command line (-D) Optional start offset and length for `incbin` Directive `filler` sets fill value (default: 0) for unused bytes Optional fill value (default: set by `filler`) for `skip` and `align`
0.12	Feb 19, 2023	Added verbose output option (-v) Created snap package for Linux (amd64) Fix: defined() checks validity of argument Exclamation mark '!' forces absolute addressing
0.11	Feb 17, 2023	Create optional listing file (-l)
0.10	Feb 15, 2023	Initial release

Disclaimer

The author, 0xC0DE, of this software accepts no responsibility for damages resulting from the use of this product and makes no warranty or representation, either express or implied, including but not limited to, any implied warranty of merchantability or fitness for a particular purpose. This software is provided "AS IS", and you, its user, assume all risks when using it.

Contact

If you have any questions, suggestions or bug reports about `max65`, please contact me at 0xC0DE6502@gmail.com or on Twitter @0xC0DE6502.