

# win2c64 manual (version 1.8)

Aart J.C. Bik (<http://www.aartbik.com/>)

## 1 Assembler Usage

This document describes the cross-assembler `win2c64`, which is an assembler that runs on any Microsoft Windows platform and generates code that can be executed on the Commodore 64. The assembler is invoked as follows

```
win2c64 [options] file.s
```

where `file.s` denotes the source file and `[options]` denotes an optional list of the following options.

Option	Meaning
-h	show short help summary
-e	envelop all segments into one
-o	show opcode table
-r	output raw bytes encoding only
-s	enable silent assembly
-u	enable undocumented opcodes
-x	show symbol table

When no errors are encountered in the source file, this command generates, by default, the target file `file.t64` in C64S tape image format [Pet]. All assembler messages are sent to `stderr`. If no source file is indicated, the assembler reads from `stdin` and writes to `stdout` to enable the use of `win2c64` in pipes and redirections.

## 2 Lexical and Syntactic Conventions

Valid assembler input consists of literal constants represented as decimal numbers (e.g. `15`), hexadecimal numbers (e.g. `$0f`), or binary numbers (e.g. `%00001111`), labels consisting of one letter followed by zero or more letters, digits, or underscores (e.g. `label` and `x_2`), mnemonics (e.g. `lda`, `inx`, and `rts`), parenthesis ( `and` ), symbols `#`, `x`, `y`, `+`, `@`, the high-byte and low-byte operators `>` and `<`, strings of characters enclosed between double quotes (e.g. `"THIS IS A STRING"`), two symbols that start a comment (`;` and `!`), and directives (`.org`, `.equ` or `=`, `.byte`, and `.align`).

Whitespace (spaces and tabs) separates tokens where required, for instance, in between a label and a mnemonic, but is otherwise ignored. A newline terminates each input line. When the assembler encounters the character ; or !, the rest of the input line is ignored as a comment. Each input line should either be empty (including comments), or otherwise define either one machine instruction or directive using the following format, where [...] denotes optional fields.

```
[label] mnemonic/directive [operand(s)] [comment]
```

The assembler is case-insensitive. This implies that, for instance, label, lAbel and Label all denote the same label. Likewise, LDA, Lda, and lda are all interpreted as the mnemonic for loading the accumulator. All mnemonics and the operands x and y are reserved keywords, which implies that they cannot be used as labels. The following input line, for example, causes an assembler error due to the invalid attempt to use the mnemonic nop as a label.

```
nop jsr chROUT ; print character
```

### 3 Machine Instructions

The assembler supports all MOS Technology 6510 machine instructions defined by combining any of the following mnemonics with the appropriate addressing formats.

```
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
```

By default, only the mnemonics shown above are recognized as valid opcodes. Under option -u, however, the assembler also recognizes the following undocumented opcodes, following the naming conventions defined in [Var96].

```
alr anc arr aso axa axs dcm hlt ins las lax lse
oal rla rra sax say skb skw tas xaa xas
```

The 6510 supports the following addressing formats.

Addressing Format	Example	Addressing Format	Example
implied	dex	absolute,x	lda \$c000,x
immediate	lda #\$ff	absolute,y	lda \$c000,y
zero page	lda \$10	(indirect,x)	lda (\$10,x)
zero page,x	lda \$10,x	(indirect),y	lda (\$10),y
zero page,y	ldx \$10,y	indirect	jmp (\$c000)
absolute	lda \$c000	relative	bne label

An error is generated when the instruction defined by the mnemonic does not support the associated addressing format. Please refer to [Com82, Smi84] for a detailed overview of all 6510 machine instructions.

## 4 Directives

The assembler supports several directives. The `org` directive takes the following format

```
[label] .org address [comment]
```

and defines that all following instructions and data must be placed starting at the defined memory address, which must be in the range `$0000` to `$ffff`. The optional label is associated with the given memory address. This directive does not place anything in memory yet. Several `org` directives may be used to break a program into non-overlapping segments.

The `equ` directive takes the format

```
label .equ value [comment]
```

or alternatively

```
label = value [comment]
```

and associates the label with the defined value, which must be in the range `$0000` to `$ffff`. The directive records the association between label and value in the symbol table for later use, but otherwise does not place anything in memory yet. Values are recorded as 2-byte constants in the symbol table, but can still be used as 1-byte operands. The assembler verifies whether all uses can be appropriately encoded. An example is shown below.

```
chROUT .equ $ffd2
one    .equ $0001
...
jsr chROUT ; use of a 2-byte value
lda #one   ; use of a 1-byte value
```

In contrast, both load instructions below are invalid, because a 2-byte value cannot be encoded as an immediate operand.

```
twobyte .equ $ffff
lda #twobyte ; invalid (.equ immediate)
lda #$ffff   ; invalid (direct immediate)
```

Since the assembler handles all data as unsigned numbers, a 1-byte representation of  $-1$  in two's complement should simply be stored as `$ff` in the symbol table, to ensure that subsequent immediate encodings proceed properly.

The `byte` directive takes the format

```
[label] .byte value-list [comment]
```

and places all following values in memory starting from the current memory address, which is associated with the optional label. The list of values can consist of any sequence of one or more whitespace- or comma-separated 1-byte literal constants, strings, and symbolic expressions that evaluate to 1-byte values. An example follows.

```

data  .byte "A STRING" $00 %0011 32 ; start of data
      .byte >data <data             ; high-byte and low-byte
                                      ; of label data itself

```

The byte directive is typically used to reserve memory for actual data, such as text output or bitmaps for screen dumps and sprites. The directive can also define executable code by simply placing the correct instruction encoding in memory, as illustrated below.

```

code  ldx  #$01 ; assembler-encoded instruction falls through
data  .byte $ea ; into user-encoded instruction of nop
      nop      ; execution continues here...

```

This directive is useful to obtain encodings that are not directly supported by the assembler. An example that encodes a BASIC line before the actual machine code is shown below.

```

      .org $0800 ; start at BASIC
      .byte $00 $0c $08 $0a $00 $9e $20 $32 ; encode SYS 2064
      .byte $30 $36 $34 $00 $00 $00 $00 $00 ; as BASIC line
main  .... ; start code here

```

When the resulting program is loaded, simply typing 'RUN' (rather than using an explicit 'SYS' to the appropriate memory address) will start executing from label `main`.

Finally, the alignment directive takes the format

```
[label] .align value [comment]
```

to obtain the indicated memory alignment, provided that the operand evaluates to any of the powers of two 1, 2, 4, ..., 256, 512, 1024. If required, the assembler pads memory with the 1-byte `nop` instruction to enforce the alignment, so that the directive can be used to enforce alignment on instruction sequences or data alike. The optional label is associated with the first memory address where this padding starts which, obviously, is not necessarily the same as the actually aligned memory address that follows. Below, an example that enforces a 64-byte alignment on the start of a loop is shown.

```

pad   .align 64 ;
loop  .... ;
      cpx #10 ;
      bne loop ; loop back

```

Here, if `pad` is not 64-byte aligned, the required number of `nop` instructions is inserted to force this memory alignment on `loop`.

## 5 Labels

Labels that are used in directives should have prior definitions. So, in the example below, all but the definition of `clow` are valid.

```
address .equ $c020      ; valid
alow    .equ <address   ; valid, alow = $20
ahig    .equ >address   ; valid, ahig = $c0
        .org address   ; valid
clow    .equ <chROUT    ; invalid, use before def
chROUT  .equ $ffd2      ; valid
```

The assembler provides a rather flexible way of defining and using labels in instructions, however. Instructions can simply refer to any label in the program, provided that eventually every used label is actually defined. So, the following example is valid.

```
        bne exit       ;
        jsr chROUT     ;
exit    rts            ; label exit   defined
chROUT .equ $ffd2     ; label chROUT defined
```

When given a choice between a 1-byte or 2-byte encoding (like absolute vs. zero page addressing), the assembler uses the smallest possible encoding when the label is already defined, or otherwise assumes the largest possible encoding which is filled in with ‘back patching’ later.

The high-byte and low-byte operators `>` and `<` yield, respectively, the most and least significant byte of its operand. For example, `>$abcd` yields `$ab` and `<$abcd` yields `$cd`. More general, operator `@` followed by one of the hexadecimal digits `0` through `f` extracts the byte that starts at the specified bit `0` through `15`, viz. `d@operand` is computed as the following logical shift right (zeros in) followed by masking the value to a byte: `(operand >> d) & $ff`. For example, `@4$abcd` yields `$bc` and `@c$abcd` yield `$0a`, while the operators `@8` and `@0` are identical to the high-byte and low-byte operator, respectively. This generalization allows for various memory address manipulations.

The `+` operator simplifies offset computations, as illustrated in the following example of self-modifying code which, each time the subroutine is called, prints a subsequent character.

```
selfmod lda #"A"       ; load  character
        jsr $ffd2     ; print character
        inc selfmod+1 ; change immediate of lda
        rts          ;
```

Adding the constant one to the label `selfmod` yields the address of the immediate field in the load instruction, which would be much harder to access without using a `+` operator.

Labels used in offset computations can still be involved in ‘back patching’, as long as at least one operand of each + operator evaluates to a constant. An example that illustrates the flexibility of this approach is shown below, where various manipulations of a yet-to-be-defined label are allowed.

```

val1   .equ $01ff           ;
val2   .equ $ff02           ;
      ldx #<later           ; becomes $cd
      ldy #>later           ; becomes $ab
      lda #@clater + >val1 + <val2 ; becomes $0d
later  .equ $abcd           ; ($0a+$01+$02)

```

Finally, to support a commonly used coding style, a single label on an otherwise empty input line (including comments) is associated with the current memory address (viz. where subsequent instructions or data would be placed). A valid example follows, where label `loop` is associated with the `jsr` instruction.

```

      ldx #99
loop
      jsr routine
      dex
      bne loop      ; loop back

```

## 6 Memory Model

The assembler supports a simple segmented memory model. By default, instructions and data are placed in the segment that starts at memory address `$c000` and up. The `originate` directive can be used to define an alternative segment, or even to break a program into several segments. The assembler reports an error if the placement of instructions or data reaches memory address `$ffff`, or if the different segments overlap.

To start executing a program, use ‘`SYS`’ to the memory address of the segment that begins with the main entry. Otherwise, query the symbol table with option `-x` to find the memory address where program execution should start. An example of a single segment program that places data before the main instructions but always starts as ‘`SYS 2048`’ is shown below.

```

initial .org $0800 ;
ijump   jmp main   ;
      ;
data    .byte "...." ; place data here
      .byte "...." ;
main    ....      ; start code here

```

Due to the semantics of the `originate` directive, both the labels `initial` and `ijump` are associated with memory address `$0800` in this example.

An example of a two segment program is shown below. The example also illustrates how to use operator @ for a memory address manipulation that determines in which 64-byte chunk of a 16K bank of memory the bitmap of a sprite resides.

```

        .org 832                                ;
        ;
bitmap  .byte %00000011 %11111111 %00000000 ; sprite bitmap
        .byte %11111111 %11111111 %11111111 ;
        :
        .byte %11111111 %11111111 %11111111 ;
        .byte %00001111 %11111111 %11110000 ;
        .org $c000                             ;
main    lda #@6 bitmap                        ; set sprite pointer
        sta $07f7                             ; to 64-byte bitmap
        :

```

Multiple segments can be combined into one enveloping segment using the option `-e`. This feature may be useful for emulators that do not easily combine multiple files into memory, at the expense of storing additional bytes in between the different segments (all set to zero).

## 7 Assembler Output Formats

The assembler converts the user readable source file `file.s` into the raw bytes encoding of the corresponding machine instructions and data which are, by default, stored as target file `file.t64` in C64S tape image format supported by Commodore 64 emulators like C64S [Pet], CCS64 [Sun], and WinVice [VIC]. The generated file defines a directory with one entry per segment that loads into the appropriate memory address.

Under option `-r`, the assembler writes each segment as raw bytes encoding only, with the first two bytes defining the initial segment address. The target file is `file.rw` for one segment, or `file.rwa`, `file.rwb`, etc. for multiple segments (this feature is less useful in pipes or redirections, since the segment encodings appear concatenated at the output).

With some effort, one of these output formats can be uploaded to a real Commodore 64 for execution. Finally, please feel free to contact the author with suggestions for other output formats.

## 8 Example

The following ‘Hello World’ program provides sample input for the cross-assembler `win2c64`.

```

chROUT .equ    $ffd2    ; kernal address
main   .org    $c000    ; start at free RAM
       ldx     #0
loop   lda     text,x
       jsr     chROUT
       inx
       cpx     #11
       bne     loop
       rts
text   .byte   "HELLO WORLD"

```

The output of applying win2c64 to this file is shown below.

```

=> win2c64 -x hello.s
**** MOS Technology 6510 Assembler
**** for the Commodore 64 (v1.8)
**** (C) 2005-2006 Aart J.C. Bik
win2c64 done in 166680 clocks
segment 0 [$c000:$c019] #bytes : 25
output format: C64S tape image

```

```

----- SYMBOL TABLE -----
chROUT                ::      $ffd2
main                  ::      $c000
loop                  ::      $c002
text                  ::      $c00e
-----

```

## References

- [Com82] Commodore Business Machines, Inc. *Commodore 64 Programmer's Reference Guide*, 1982.
- [Pet] Miha Peternel. *C64S Commodore 64 Emulator for PC*. See <http://www.phs-edv.de/c64s/index.htm>.
- [Smi84] Bruce Smith. *Commodore 64 Assembly Language*. Shiva Publishing, 1984.
- [Sun] Per Håkan Sundell. *CCS64 Commodore 64 Emulator*. See <http://www.ccs64.com/>.
- [Var96] Adam Vardy. *Extra Instructions Of The 65XX Series CPU*, 1996.
- [VIC] VICE Team. *WinVICE Emulator*. See <http://www.viceteam.org/>.