WS Assembler Description

Based on
UNIX Assembler Reference Manual
by Dennis M. Ritchie

0. Introduction

This document describes the intended usage, and language accepted by the 6502 assembler I will be writing. In this document, the assembler to be written will be referred to as the "WS assembler".

The WS assembler is a two-pass assembler without macro capabilities. By default, it produces an output file which contains relocation information and a complete symbol table. The output, therefore, is suitable for linking by an as-yet nonexistent linker, which may be used to combine the outputs of several assembler runs and to obtain object programs from libraries. The WS assembler will, however, also be capable of producing an absolute binary which is executable without further processing.

1. Usage

The WS assembler is intended to be run from within the ORCA command shell on an Apple II host computer. The ORCA command shell provides programs with the ability to receive command line arguments. The WS assembler accepts several command line switches to control its behavior.

The "-a" switch specifies that the assembler should produce an absolute binary. The "-a" switch may be followed by an address specifying the loading address for the output binary. If no loading address is given, the address $2000 (8192) will be assumed. (See **.abs** directive §7.9)

The "-o" switch allows the user to specify a name for the output file produced by the assembler. If the "-o" switch is not given, the name of the output file will be "a.out". The resulting output file, if any, is placed in the current directory.

Any command line arguments apart from valid switches are assumed to be the names of 6502 assembly language source files. The files specified by these arguments are concatenated and assembled. Thus programs may be written in several pieces and assembled together.

2. Lexical conventions

Assembler tokens include identifiers (alternatively, ‘‘symbols’’ or ‘‘names’’), temporary symbols, constants, and operators.

2.1 Identifiers

An identifier consists of a sequence of alphanumeric characters (including period " . " and underscore "_" as alphanumeric) of which the first may not be numeric. Only the first eleven characters are significant.

2.2 Temporary symbols

A temporary symbol consists of a digit followed by "f " or "b". Temporary symbols are discussed fully in §5.1.

2.3 Constants

Constants yield a numerical value of either 8- or 16-bits length as described below:

2.3.1 Decimal constants

A decimal constant consists of a sequence of one or more decimal digits. The magnitude of the constant should be representable in 15 bits; i.e., be less than 32,768. A decimal constant in the range −128 to +127 yields an 8-bit value unless it is prefixed by the @ operator, which forces the constant to 16-bit width. Values less than −128 and greater than +127 yield a 16-bit value unless prefixed by the < or > operators, which force the constant to 8-bit width. (see §6.1, Expression operators, below).

2.3.2 Hexadecimal constants

A hexadecimal constant consists of a dollar sign "$", followed by a sequence of one or more hexadecimal digits, which include the decimal digits "0" through "9", as well the letters "A" through "F" in either upper- or lowercase. The constant should be representable in 16 bits; i.e., be less than or equal to $FFFF. A hexadecimal constant consisting of one or two hexadecimal digits yields and 8-bit value, unless it is prefixed by the @ operator, which forces the constant to 16-bit width. A hexadecimal constants consisting of more than two hexadecimal digits yields a 16-bit value unless prefixed by the < or > operators, which force the constant to 8-bit width. (see §6.1, Expression operators, below).

2.3.3 Character constants

A character constant consists of a single quote character followed by an ASCII character, which is not a newline characters, followed by another single quote character. Certain dual-character escape sequences are acceptable in place of the ASCII character to represent newline and other non-graphic characters (see String statements, §5.5). A character constant yields an 8-bit value unless it is prefixed by the @ operator, which forces the constant to 16-bit width.

2.4 Operators

There are several single- and double-character operators; see §6.

2.5 Blanks

Blank and tab characters may be interspersed freely between tokens, but may not be used within tokens (except character constants). A blank or tab is required to separate adjacent identifiers or constants not otherwise separated.

2.6 Comments

The character ";" introduces a comment, which extends through the end of the line on which it appears. Comments are ignored by the assembler.

3. Segments

Assembled code and data fall into three segments: the text segment, the data segment, and the bss segment. The text segment is the one in which the assembler begins, and it is the one into which instructions are typically placed.

The data segment is available for placing data or instructions which will be modified during execution. Anything which may go in the text segment may be put into the data segment.

The bss segment may not contain any explicitly initialized code or data. The length of the bss segment (like that of text or data) is determined by the high-water mark of the location counter within it. The bss segment is actually an extension of the data segment and begins immediately after it. At the start of execution of a program, the bss segment is set to 0. Typically the bss segment is set up by statements exemplified by

```
lab: . = .+10
```

The advantage in using the bss segment for storage that starts off empty is that the initialization information need not be stored in the output file. See also *Location counter* and *Assignment statements* below.

4. The location counter

One special symbol, ".", is the location counter. Its value at any time is the offset within the appropriate segment of the start of the statement in which it appears. The location counter may be assigned to, with the restriction that the current segment may not change; furthermore, the value of "." may not decrease. If the effect of the assignment is to increase the value of ".", the required number of null bytes are generated (but see *Segments* above).

5. Statements

A source program is composed of a sequence of statements. Statements are separated either by new-lines or by commas. There are five kinds of statements: null statements, expression statements, assignment statements, string statements, and keyword statements.

Any kind of statement may be preceded by one or more labels.

5.1 Labels

There are two kinds of label: name labels and numeric labels. A name label consists of a name followed by a colon. The effect of a name label is to assign the current value and type of the location counter "." to the name. An error is indicated in pass 1 if the name is already defined; an error is indicated in pass 2 if the "." value assigned changes the definition of the label.

A numeric label consists of a digit 0 to 9 followed by a colon. Such a label serves to define temporary symbols of the form "*n* b" and "*n* f", where *n* is the digit of the label. As in the case of name labels, a numeric label assigns the current value and type of "." to the temporary symbol. However, several numeric labels with the same digit may be used within the same assembly. References of the form "*n* f" refer to the first numeric label "*n*:" f orward from the reference; "*n*b" symbols refer to the first "*n*:" label backward from the reference. This sort of temporary label was introduced by Knuth [The Art of Computer Programming, Vol I: Fundamental Algorithms]. Such labels tend to conserve both the symbol table space of the assembler and the inventive powers of the programmer.

5.2 Null statements

A null statement is an empty statement (which may, however, have labels). A null statement is ignored by the assembler. Common examples of null statements are empty lines or lines containing only a label.

5.3 Expression statements

An expression statement consists of an arithmetic expression not beginning with a keyword. The assembler computes its value and, depending on the width of the resulting expression, places one or two bytes in the output stream, together with the appropriate relocation bits. The rules for determining the width of an expression are as follows: Constants can be either 8- or 16-bits in width. Relative symbols, such as addresses are always 16 bits wide. When combining subexpressions of different widths, the result is always 16 bits wide. The most common way to specify the desired width of an expression is by using the <, >, and @ operators at the highest level of the expression. (see §6.1, Expression operators, below).

5.4 Assignment statements

An assignment statement consists of an identifier, an equals sign, and an expression. The value and type of the expression are assigned to the identifier. It is not required that the type or value be the same in pass 2 as in pass 1, nor is it an error to redefine any symbol by assignment.

Any external attribute of the expression is lost across an assignment. This means that it is not possible to declare a global symbol by assigning to it, and that it is impossible to define a symbol to be offset from a non-locally defined global symbol.

As mentioned, it is permissible to assign to the location counter ".". It is required, however, that the type of the expression assigned be of the same type as ".", and it is forbidden to decrease the value of ".". In practice, the most common assignment to "." has the form ". = . + *n*" for some number *n*; this has the effect of generating *n* null bytes.

5.5 String statements

A string statement generates a sequence of bytes containing ASCII characters. A string statement consists of a left double quote followed by a sequence of ASCII characters not including newline, followed by a closing double quote. Any of the ASCII characters may be replaced by a two-character escape sequence to represent certain non-graphic characters, as follows:

Sequence	ASCII Character	Numeric Value
\n	NL	10
\t	HT	9
\0	NUL	0
\r	CR	13
\\	\	
\"	"	
\'	'	

The last two are included so that the escape character and quote characters may be represented. The same escape sequences may also be used within character constants (see §2.3 above).

5.6 Keyword statements

Keyword statements are numerically the most common type, since most machine instructions are of this sort. A keyword statement begins with one of the many predefined keywords of the assembler. The syntax of the remainder depends on the keyword. All the keywords are listed below with the syntax they require.

6. Expressions

An expression is a sequence of symbols representing a value. Its constituents are identifiers, constants, temporary symbols, operators, and parentheses. Each expression has a type.

There are both unary and binary operators. Arithmetic is two's complement and has 16 bits of precision. Operators have varying precedences, and operators of higher precedence are applied before operators of lower precedence. When multiple operators of the same precedence occur in sequence in an expression or subexpression, the expression or subexpression is evaluated from left to right.

6.1 Expression operators

The operators are, in order of precedence from high to low:

+ − ~	unary plus, unary minus, bitwise complement
< > @	force the width of the subexpression to 8-bits, with the value being the least significant byte of the subexpression value, force the width of the subexpression to 8-bits, with the value being the most significant byte of the subexpression value, force the width of the subexpression to 16-bits.
* / % << >>	multiplication, integer division, integer modulo, logical left shift, logical right shift
+ −	addition, subtraction
&	bitwise and
| ^	bitwise or, bitwise exclusive or
!	result has the value of first operand and the type of the second

Expressions may be grouped by use of parentheses.

6.2 Types

The assembler deals with a number of types of expressions. Most types are attached to keywords and used to select the routine which treats that keyword. The types likely to be met explicitly are:

undefined

> Upon first encounter, each symbol is undefined. It may become undefined if it is assigned an undefined expression. It is an error to attempt to assemble an undefined expression in pass 2. In pass 1, it is not an error (except that certain keywords require operands which are not undefined).

undefined external

> A symbol which is declared **.extern** but not defined in the current assembly is an undefined external. If such a symbol is declared, the linker must be used to load the assembler's output with another routine that defines the undefined reference.

absolute

An absolute symbol is one defined ultimately from a constant. Its value is unaffected by any possible future applications of the linker to the output file. Values of symbols in absolute assemblies are always absolute (see **.abs** directive §7.9).

text

The value of a text symbol is measured with respect to the beginning of the text segment of the program. If the assembler output is linked, its text symbols may change in value since the program need not be the first in the linker's output. Most text symbols are defined by appearing as labels. At the start of an assembly, the value of "." is text 0.

data

The value of a data symbol is measured with respect to the origin of the data segment of a program. Like text symbols, the value of a data symbol may change during subsequent linking since previously loaded programs may have data segments. After the first **.data** statement, the value of "." is data 0.

bss

The value of a bss symbol is measured from the beginning of the bss segment of a program. Like text and data symbols, the value of a bss symbol may change during subsequent linking, since previously loaded programs may have bss segments. After the first **.bss** statement, the value of "." is bss 0.

external absolute, text, data, or bss

symbols declared **.extern** but defined within an assembly as absolute, text, data, or bss symbols may be used exactly as if they were not declared **.extern,** however, their value and type are available to the link editor so that the program may be loaded with others that reference these symbols.

other types

Each keyword known to the assembler has a type which is used to select the routine which processes the associated keyword statement. The behavior of such symbols when not used as keywords is the same as if they were absolute.

Values of type absolute, text, data, and bss, whether external or not, can additionally be either 16-bit or 8-bit values. When a 8-bit value is of non-absolute type, its type also includes the information whether it represents the least significant or most significant byte of a relocatable 16-bit value.

6.3 Type propagation in expressions

When operands are combined by expression operators, the result has a type which depends on the types of the operands and on the operator. The rules involved are intended to be sensible and predictable. For purposes of expression evaluation the important types are

undefined
absolute
text
data
bss
undefined external
other

The combination rules are: If one of the operands is undefined, the result is undefined. If both operands are absolute, the result is absolute. If an absolute is combined with one of the "other" types mentioned above, or with a register expression, the result has the other type. An "other" type combined with an explicitly discussed type other than absolute acts like an absolute.

Further rules applying to particular operators are:

+	If one operand is text-, data-, or bss-segment relocatable, or is an undefined external, the result has the postulated type and the other operand must be absolute.
−	If the first operand is a relocatable text-, data-, or bss-segment symbol, the second operand may be absolute (in which case the result has the type of the first operand); or the second operand may have the same type as the first (in which case the result is absolute). If the first operand is external undefined, the second must be absolute. All other combinations are illegal.
!	This operator follows no other rule than that the result has the value of the first operand and the type of the second.
others	It is illegal to apply these operators to any but absolute symbols.

7. Directives

The keywords listed below introduce statements which generate data in unusual forms or influence the later operations of the assembler. The metanotation

[stuff] ...

means that 0 or more instances of the given stuff may appear. Also, boldface tokens are literals, italic words are substitutable.

7.1 **.byte** *expression* [, *expression*] ...

The expressions in the comma-separated list are truncated to 8 bits and assembled in successive bytes. This is an alternative to assembling 8-bit values as expression statements using the < operator

7.2 **.addr** *expression* [, *expression*] ...

The expressions in the comma-separated list are assembled as 16-bit values in little-endian order. This is an alternative to assembling 16-bit values as expression statements.

7.3 **.if** *expression*

The expression must be absolute and defined in pass 1. If its value is nonzero, the **.if** is ignored; if zero, the statements between the **.if** and the matching **.else** or **.endif** (below) are ignored. **.if** may be nested. The effect of **.if** cannot extend beyond the end of the input file in which it appears.

7.4 **.else**

This statement marks the end of section of code prefixed by **.if** and begins a section of code that will be assembled if the expression accompanying the preceding **.if** pseudo-op had a value of zero.

7.5 **.endif**

This statement marks the end of a conditionally-assembled section of code. See **.if** above.

7.6 **.extern** *name* [, *name*] ...

This statement makes the names external. If they are otherwise defined (by assignment or appearance as a label) they act within the assembly exactly as if the **.extern** statement were not given; however, the linker may be used to combine this routine with other routines that refer to these symbols.

Conversely, if the given symbols are not defined within the current assembly, the linker can combine the output of this assembly with that of others which define the symbols.

7.6 **.text**
7.7 **.data**
7.8 **.bss**

These three directives cause the assembler to begin assembling into the text, data, or bss segment respectively. Assembly starts in the text segment. It is forbidden to assemble any code or data into the bss segment, but symbols may be defined and "." moved about by assignment.

7.9 **.abs** *expression*

The **.abs** directive must be given before the value of the location counter "." has changed in any segment. It causes the assembler to generate an absolute (non-relocatable) assembly of the source file. The expression following the **.abs** directive specifies the initial address for the location counter "." of the text segment. The **.abs** directive is an alternative to the -a command line switch described in §1.

7.10 **.6502**

Disable the recognition and assembly of 65c02 instructions

7.11 **.65c02**

Enable the recognition and assembly of 65c02 instructions

8. Machine instructions

Because of the rather complicated instruction and addressing structure of the 6502, the syntax of machine instruction statements is varied.

8.1 Simple machine instructions

The following instructions are defined as absolute symbols:

clc	**php**		**sed**
cld	**phx**	(65c02)	**sei**
cli	**phy**	(65c02)	**tax**
clv	**pla**		**tay**
dex	**plp**		**tsx**
dey	**plx**	(65c02)	**txa**
inx	**ply**	(65c02)	**txs**
iny	**rti**		**tya**
nop	**rts**		
pha	**sec**		

They therefore require no special syntax.

8.2 Branch

The following instructions take an expression as operand. The expression must lie in the same segment as the reference, cannot be undefined-external, and its value cannot be more then 126 bytes less than, nor more than 129 bytes greater thatn the current value of the location counter.

bcc	*expression*	
bcs	*expression*	
beq	*expression*	
bmi	*expression*	
bne	*expression*	
bpl	*expression*	
bra	*expression*	(65c02)
bvc	*expression*	
bvs	*expression*	

8.3 Extended branch

The following symbols are followed by an expression representing an address in the same segment as ".". If the target address is close enough, a single branch instruction is generated. Otherwise the converse branch over a jmp to the target address is generated.

jcc *expression*
jcs *expression*
jeq *expression*
jmi *expression*
jne *expression*
jpl *expression*
jvc *expression*
jvs *expression*

8.5 **trb** and **tsb** instructions (65c02)

The symbols **trb** and **tsb** are followed by an operand as shown below. If the operand has a value less than 256, the expression is interpreted as an 8-bit reference to the 6502 zero page and assembled accordingly. Otherwise the expression is interpreted as a reference to a 16-bit address and assembled accordingly. It is possible to assemble the expression as a reference to a 16-bit address even when it has a value less than 256 by use of the @ operator.

trb	*# expression*	;immediate operand	(65c02)
tsb	*expression*	;address operand	(65c02)

8.5 **cpx** and **cpy** instructions

The symbols **cpx** and **cpy** are followed by an operand as shown below. If the expression in an immediate operand has a 16-bit value, it is truncated to 8 bits. If the expression in an address (non-immediate) operand has a value less than 256, the expression is interpreted as an 8-bit reference to the 6502 zero page and assembled accordingly. Otherwise the expression is interpreted as a reference to a 16-bit address and assembled accordingly. It is possible to assemble the expression as a reference to a 16-bit address even when it has a value less than 256 by use of the @ operator.

cpx	*# expression*	;immediate operand
cpy	*expression*	;address operand

8.6 Shift, **dec** and **inc** instructions

The symbols **asl**, **lsr**, **rol**, **ror**, **dec** and **inc** are followed by an operand as shown below. If the expression in an operand has a value less than 256, the expression is interpreted as an 8-bit reference to the 6502 zero page and assembled accordingly. Otherwise the expression is interpreted as a reference to

a 16-bit address and assembled accordingly. It is possible to assemble the expression as a reference to a 16-bit address even when it has a value less than 256 by use of the @ operator.

```
asl                     ;accumulator operand        (dec, inc: 65c02)
lsr     a               ;accumulator operand        (dec, inc: 65c02)
dec     expression      ;address operand
inc     expression, x   ;X-indexed operand
```

8.7 **ldx** instruction

The symbol **ldx** is followed by an operand as shown below. If the expression in an immediate operand has a 16-bit value, it is truncated to 8 bits. If the expression in a non-immediate operand has a value less than 256, the expression is interpreted as an 8-bit reference to the 6502 zero page and assembled accordingly. Otherwise the expression is interpreted as a reference to a 16-bit address and assembled accordingly. It is possible to assemble the expression as a reference to a 16-bit address even when it has a value less than 256 by use of the @ operator.

```
ldx     # expression    ;accumulator operand
ldx     expression      ;address operand
ldx     expression, y   ;Y-indexed operand
```

8.8 **bit** and **ldy** instructions

The symbols **bit** and **ldy** are followed by an operand as shown below. If the expression in an immediate operand has a 16-bit value, it is truncated to 8 bits. If the expression in a non-immediate operand has a value less than 256, the expression is interpreted as an 8-bit reference to the 6502 zero page and assembled accordingly. Otherwise the expression is interpreted as a reference to a 16-bit address and assembled accordingly. It is possible to assemble the expression as a reference to a 16-bit address even when it has a value less than 256 by use of the @ operator.

```
ldy     # expression    ;accumulator operand        (bit: 65c02)
ldy     expression      ;address operand
ldy     expression, x   ;X-indexed operand          (bit: 65c02)
```

8.9 **stx** instruction

The symbol **stx** is followed by an operand as shown below. If the expression in an address operand has a value less than 256, the expression is interpreted as an 8-bit reference to the 6502 zero page and assembled accordingly. Otherwise the expression is interpreted as a reference to a 16-bit address and assembled accordingly. It is possible to assemble the expression as a reference to a 16-bit address even when it has a value less than 256 by use of the @ operator. The expression in a Y-indexed operand must have a value less than 256, as it is interpreted as a reference to the 6502 zero page.

```
stx     expression      ;address operand
stx     expression, y   ;Y-indexed operand
```

8.10 **sty** and **stz** instructions

The symbols **sty** and **stz** are followed by an operand as shown below. If the expression in an address operand has a value less than 256, the expression is interpreted as an 8-bit reference to the 6502 zero page and assembled accordingly. Otherwise the expression is interpreted as a reference to a 16-bit address and assembled accordingly. It is possible to assemble the expression as a reference to a 16-bit address even when it has a value less than 256 by use of the @ operator. The expression in an X-indexed operand must have a value less than 256, as it is interpreted as a reference to the 6502 zero page.

```
sty     expression          ;address operand            (stz: 65c02)
sty     expression, x       ;X-indexed operand          (stz: 65c02)
```

8.11 **jmp** instruction

The symbol **jmp** is followed by an operand as shown below. The expression is interpreted as a reference to a 16-bit address and assembled accordingly.

```
jmp     expression          ;address operand
jmp     (expression)        ;indirect operand
jmp     (expression, x)     ;X-indexed indirect operand     (65c02)
```

8.12 **jsr** instruction

The symbol **jsr** is followed by an operand as shown below. The expression is interpreted as a reference to a 16-bit address and assembled accordingly.

```
jsr     expression          ;address operand
```

8.13 **sta** instruction

The symbols **sta** is followed by an operand as shown below. If the expression in a address operand or X-indexed operand has a value less than 256, the expression is interpreted as an 8-bit reference to the 6502 zero page and assembled accordingly. Otherwise the expression is interpreted as a reference to a 16-bit address and assembled accordingly. It is possible to assemble the expression as a reference to a 16-bit address even when it has a value less than 256 by use of the @ operator. The expression in a Y-indexed operand is interpreted as a reference to a 16-bit address and assembled accordingly. The expression in a zero page indirect operand, an X-indexed zero page indirect operand, or in a zero page indirect Y-indexed operand must have a value less than 256, as it is interpreted as a reference to the 6502 zero page.

```
sta     expression          ;address operand
sta     expression, x       ;X-indexed operand
```

```
sta     expression, y       ;Y-indexed operand
sta     (expression)        ;zero page indirect operand              (65c02)
sta     (expression, x)     ;X-indexed zero page indirect operand
sta     (expression), y     ;zero page indirect Y-indexed operand
```

8.14 Arithmetic and logical instructions

The symbols **adc**, **and**, **cmp**, **eor**, **lda**, **ora** and **sbc** are followed by an operand as shown below. If the expression in an immediate operand has a 16-bit value, it is truncated to 8 bits. If the expression in a address operand or X-indexed operand has a value less than 256, the expression is interpreted as an 8-bit reference to the 6502 zero page and assembled accordingly. Otherwise the expression is interpreted as a reference to a 16-bit address and assembled accordingly. It is possible to assemble the expression as a reference to a 16-bit address even when it has a value less than 256 by use of the @ operator. The expression in a Y-indexed operand is interpreted as a reference to a 16-bit address and assembled accordingly. The expression in a zero page indirect operand, an X-indexed zero page indirect operand, or in a zero page indirect Y-indexed operand must have a value less than 256, as it is interpreted as a reference to the 6502 zero page.

```
adc     # expression       ;accumulator operand
and     expression         ;address operand
cmp     expression, x      ;X-indexed operand
eor     expression, y      ;Y-indexed operand
lda     (expression)       ;zero page indirect operand              (65c02)
ora     (expression, x)    ;X-indexed zero page indirect operand
sbc     (expression), y    ;zero page indirect Y-indexed operand
```

8.15 **brk** instruction

The symbols **brk** is optionally followed by a single expression, which must be absolute. If the expression is not provided, a value of zero is assumed. The value of the expression is truncated to 8 bits and inserted into the output stream following the value of the **brk** instruction itself. The symbol **sys** is a synonym for **brk**.

```
brk                        ;implied operand with value zero
sys     expression         ;explicit operand
```

9. Partial Absolute Assembly

It is sometimes useful to assemble part of a source file as an absolute assembly, even though the source is assembled in a relocatable manner. For example, if a piece of code were intended to be copied to a specific address and executed there. This can be done by assigning the starting address of the absolute code to the location counter "." as an absolute value. This sets the value of the location counter, but also changes its type to absolute, causing all further output in the current segment to be non-relocatable. The section of absolutely assembled code and data can be ended, and the generation of relocatable output resumed, bu assigning the symbol ".." to the location counter. The symbol ".." is similar to the location

counter, but it cannot be assigned to, and its value is always the current offset in the current segment. Thus, assigning ".." to "." restores the relocatable value and type of "." to what it would be, if the absolutely assembled code had been assembled as relocatable code. An example follows:

```
            .text
copyme:                 ; this code is to be copied to $300
                        ;    and executed there
.=$300                  ; begin absolute assembly
            cmp   #0
            bcc   1f
            inx
            jmp   2f
1:          iny
2:          rts
.=..                    ;go back to relocatable assembly
copylen=.-copyme
```

10. Diagnostics

When an input file cannot be read, a message to that effect is output and assembly ceases. When syntactic or semantic errors occur, a diagnostic is printed with the line number and the file name in which it occurred. Errors in pass 1 cause cancellation of pass 2. The possible errors are:

Expected (
: A left parenthesis was expected in the input (for example, as part of an addressing mode), but not found.

Expected)
: A right parenthesis was expected in the input (for example, as part of an addressing mode), but not found.

Expected ,
: A comma was expected in the input (for example, as part of an addressing mode), but not found.

Expected index register
: An index register specification (**x** or **y**) was expected in the input (for example, as part of an addressing mode), but not found.

Expected 8-bit value
: A 16-bit value was found where an 8-bit value was expected, such as the operand of an instruction using immediate addressing.

Expected 16-bit value
: An 8-bit value was found where a 16-bit value was expected, such as the operand of an instruction using absolute addressing.

Bad addressing mode
: The addressing mode specified is illegal for the instruction for which it is specified (for example, "`beq label,x`").

Syntax error in statement
: The statement is not a valid statement of one of the types described in §5. This is usually an unrecognized instruction or directive.

65c02 instruction
: A 65c02 specific instruction or addressing mode was found while assembling in 6502 mode.

Unterminated character constant
: A newline or end-of-file was found after the opening single quote but before the closing single quote of a character constant.

Unterminated string
: A newline or end-of-file was found after the opening double quote but before the closing double quote of a string constant.

Unterminated .if
: An end-of-file was found after an **.if** directive but before the corresponding **.endif** directive.

Illegal assignment to "."
: An assignment to the location counter "." decreased its value.

Branch address out of range
: An operand to a branch instruction (not including extended branch instructions) resolves to an address outside the range of an 8-bit signed relative offset. Specifically, the target address is more than 126 bytes less than, or more than 129 bytes greater than the current value of the location counter ".".

Bad temporary label
: A reference was made to a temporary (numeric) label that does not exist. For example, a referene to **1b** at the beginning of a source file before the label **1:** has been seen.

Bad character
: A control character or other illegal or unknown character has been found in the source file.

Symbol redefined as label
: A symbol that has already been defined is used as a label. This error will occur no matter how the symbol was previously defined, whether the previous definition was as a label or not.

Inconsistent location counter
: The value of a relocatable symbol is different in pass 2 than it was in pass 1.

Undefined symbol

An undefined, non-external symbol has been encountered in pass 2.