ASSEMBLER / MONITOR 64

Powerful 6510 MACRO Assembler
Development Package
for the Commodore 64

By: Lothar Englisch

A Data Becker Product

First Publishing Ltd Unit 20B, Horseshoe Road
Horseshoe Park
Pangbourne, Berks
Tel: 07357 5244

COPYRIGHT NOTICE

WARRANTY

First Printing: September 1984
Printed in U.S.A. Translated by Greg Dykema

ISBN 0 948015 06 3

This edition typeset and printed by
Quorum Technical Services Ltd., Cheltenham
November 1984

Table of Contents

THE ASSEMBLER

ASSEMBLER 64 is a two-pass 6510 or 6502 assembler for the Commodore 64. It is written entirely in machine language and occupies 8K bytes of RAM. It allows free-form input using the builtin BASIC editor, produces complete assembly listings, loadable symbol tables, various options for storing created object codes, redefinable symbols, and a comprehensive set of pseudo-ops (assembler directives) for such things as creating macros or conditional assembly. The syntax for the most part adheres to the MOS standard.

A. Using ASSEMBLER 64

ASSEMBLER 64 is loaded from diskette and requires 8K of the BASIC RAM. (address $6000-$9FFF). The area most frequently used for machine language programs from $C000 to $CFFF is left free and can be used for MONITOR 64 ($C000-$CBFF) or your own machine language programs.

Loading ASSEMBLER 64

Insert the ASSEMBLER/MONITOR distribution diskette and type:

```
LOAD "ASSEMBLER 64",8,1
```

The following appears on the screen:

```
LOADING

ASSEMBLER 64 V2.0 IS LOADING ...

*** ASSEMBLER 64 V2.0 ***
(C) 1984 DATA BECKER GMBH
2
0000-0000
NO ERRORS
READY.
```

When loading, ASSEMBLER 64 protects itself from being overwritten by BASIC. You are left with 30717 bytes for your assembly language source programs.

The **2** in the message indicates the start of pass 2. Following is the address range of the created object code and the number of errors.

Assembler programs are entered using line numbers just like BASIC programs. Lines can be changed, deleted, or inserted exactly as in BASIC. No other editor is necessary and more storage space is available for your

source programs - a total of 30K. You can separate several assembler commands on the same line using colons as in BASIC.

You can make your assembly language programs easier to read by placing an up arrow as the first character of a line. After this, all spaces are accepted and the arrow is ignored by ASSEMBLER 64. This allows you to indent your programs as desired.

ASSEMBLER 64 uses almost the same source format as the MOS standard. If even you are familiar with this standard, you should read this description because it also explains the departures from the MOS standard. The examples illustrate the instructions.

This manual is not intended to teach 6510 assembly language programming. We recommend other books such as *The Machine Language Book for the Commodore 64* or the *Advanced Machine Language for the Commodore 64* for more information on the use of macros and floating-point arithmetic.

Lines of ASSEMBLER 64 source code consist of labels, instruction mnemonics, the operands, and comments. In addition, there are several "pseudo-ops," which are not machine language instructions but rather tell the assembler to do special things. These pseudo-ops are described later in the manual.

Each program line contains a mnemonic or pseudo-op and may begin with a label (symbol). If a line is supposed to contain a label, simply place it in front of the instruction, followed by one or more spaces. A label must begin with a letter followed by other letters, numbers or periods. The first 8 characters of a label must be unique (that is, no labels may have the same first 8 characters). Non-alphanumeric characters are not allowed.

Instruction mnemonics may follow a label or may begin at the start of a line if no label is present. All mnemonics consist of 3 letters. Mnemonics are reserved words and may not be used as labels.

If an instruction begins with a period ("."), it is treated as a pseudo-op. There are three pseudo-ops which do not begin with a period. All pseudo-ops must be separated from their operands by spaces, with the exception of "=" and "*=". Pseudo-ops which begin with a period are distinguished by the first three characters only, although they will be printed in full in the assembly listing.

A line can be terminated by a semicolon. Everything following the semicolon is ignored by the assembler and can contain comments. Comments are printed out in the assembly listing but are otherwise disregarded. A colon within a comment ends it and begins a new instruction, as long as the colon is not placed within quotation marks.

If a line begins with a semicolon, the assembler treats the entire line as a comment. Such lines are printed without a line number.

The operand field contains the addressing mode and an expression for the command or pseudo-op. A semicolon may follow.

The addressing modes with expressions have the following syntaxes:

#expression	absolute addressing
expression	absolute or relative addressing
expression,x	absolute,x indexed by x
expression,y	absolute,y indexed by y
(expression,x)	indexed indirect addressing
(expression),y	indirect indexed addressing
(expression)	indirect addressing

ASSEMBLER 64 automatically converts absolute addressing to zero-page addressing if the expression has a value less than 256. If you want to force absolute addressing, you can place an exclamation point in front of the expression. LDA !5,X creates the code BD 05 00, the absolute form of LDA, while LDA 5,X yields the zero-page addressing B5 05. This is useful if you want to avoid the wrap-around effect of indexed addressing with addresses under 256.

B. Expressions

ASSEMBLER 64 is unique among assembler in its ability to calculate complex expressions. The assembler has a recursive routine for calculating nested expressions, which gives you more capabilities than other assemblers. An ASSEMBLER 64 expression may be placed wherever the word "expression" appears in a list. Such an expression is also allowed for the pseudo-ops which expect a numerical argument. The expression evaluation of ASSEMBLER 64 is so efficient that your programs can be written entirely using symbols. This makes changing and transporting ASSEMBLER 64 programs especially simple and easy to understand.

The syntax of expressions is very simple and is a superset of the MOS standard. Expressions are entered exactly as they would be on a pocket calculator which does not use an algebraic evaluation system but does allow parentheses. All operators are evaluated strictly from left to right, although square brackets are allowed as well as parentheses in order to alter the order of evaluation.

An expression can be terminated by a variety of characters. The end of a line always ends an expression. Colons, semicolons, and commas also end an expression, provided that these are not enclosed in quotation marks. A closing parenthesis ends an expression provided no unpaired open parenthesis remain. This makes nested expressions possible with indexed addressing.

You can use the following operators in expressions:

+	add values
–	subtract right value from left value
*	multiply values
/	divide left value by right value
!	logical OR of two values
&	logical AND of two values
^	logical XOR (exclusive or) of two values
>	shift left argument as many bits to the right as the right argument specifies
<	shift left argument as many bits to the left as the right argument specifies

All operations are performed using 16 bit arithmetic, although various operations will lead to overflows, such as multiplication by a value greater than 32767, or shifting left more than 15 bits. These cause an ILLEGAL QUANTITY ERROR. This error message also appears for a division by zero. For addition and subtraction, a result greater than 65535 is interpreted as a negative number in two's complement form.

The operands themselves can appear in a variety of forms. In the following the syntax is given together with an example.

Operand types

Type	Example	Syntax
hexadecimal	$1C3	$(hexdigit)
decimal	127	(digit)
binary	%110011	%(0 or 1)
PC	*	
ASCII character	"A"	"character"
label	SYMB	alphabetic(alphanumeric)
expression	("Z"+6)	(expression)

Under "Syntax," items placed within braces {} may be repeated as often as necessary.

Each of the above terms can be combined with the previously-described operators. These can be enclosed in parentheses as desired in order to alter the order of evaluation. A minus sign can be placed in front of every operand, including parenthesized expressions, to yield a two's complement value.

An entire expression can be changed by a single modifying character. One example is the use of ! to select an absolute addressing mode. In addition, the "greater than" and "less than" signs are allowed. ">" in front of expressions tells the assembler to take only the most significant byte of the expression's result (first 8 bits of the 16 bit expression), while "<" denotes the least significant byte. This is necessary for direct addressing or with the .BYTE pseudo-op. The most significant byte operator (>) performs the same operation as:

```
expression > 8
```

The least significant operator can also be represented as:

```
expression & $FF
```

Sample expressions

```
>LABEL-1+(TABLE*2)
VALUE-*
"0"-"A" < 3 + ("D" - "A" > 2&%111)
```

Parentheses may be nested as deep as necessary. Modifiers cannot be used on parenthesized parts of expressions.

C. Pseudo-ops

Most ASSEMBLER 64 pseudo-ops begin with a period (.). All of these "period" opcodes must be separated from following characters by at least one space. In addition, there are three special pseudo-ops which are defined by special characters. Pseudo-ops are recognized by their first three letters; everything else up to the next space will be ignored, although it will be printed in the listing.

The three special pseudo-ops serve to define symbols and the program counter.

1. Symbol value assignment

The simplest of these is the operator for symbol definition, the equal sign (=). In order to assign a value (expression) to a symbol, you simply write:

```
symbol = expression
```

The assignment is made only during pass 1 of the assembly. Any subsequent definition of this same symbol in the source program results in a "REDEFINITION ERROR." The "=" sign is used to define constants and addresses in symbolic form, so that only one line need be changed to alter all occurrences of the value. Here's a few examples:

```
BEGIN = $C000 ;define start of program
TAPEBUF = 828 ;define tape buffer at $33C
```

2. Redefining symbol values

Similar to the operator for symbol definition is the assignment operator, which is written as a left arrow (<–) and is used with the same syntax:

```
symbol <– expression
```

By contrast to the previous operator, it is possible to redefine a symbol. In this case, the assignment is made during pass 2 as well as pass 1. This can be used for various purposes, most often during conditional assembly (see .GOTO). Here are some examples:

```
NUMBER <– NUMBER – 1 ; decrement value
PROGRAM <– *
```

3. Program counter assignment

The third special pseudo-op controls the program counter. It is written as *= which means "assign a value to the program counter". The primary use of

this symbol is to specify the starting address of the program. If not specified, it defaults to $C000.

Storage for data may also be reserved. The statement *=*+32, for example, defines a 32-byte block beginning at the current program counter location. The value of the program counter is then incremented by 32. If a symbol is found in the label field, the value of the program counter is assigned to it **before** the program counter is incremented. Here's an example that defines variable in page 1.

```
*= $200              ; sets the program counter to the start of page 1
ADDRESS *= *+1       ; a one-byte address, set to zero
TABLE *= *+32        ; table begins at $201
LABEL *= *+1         ; LABEL has the value $233
TWO *= *+2           ; two-byte pointer
TEST *= $800         ; TEST has the value $236;
                         following code begins at $800
```

To define a table within a program, the following can be used:

```
                     ...
                     LDA #5
                     RTS
TABLE                *= *+256 ; 256-BYTE TABLE
TEST                 LDA #>ADDRESS*3
                     ...
```

In general, you can use *= to define symbols by altering the program counter. You should not, however, move it backwards. This is allowed only: 1) if you assemble object code directly into memory and execute it there; or 2) when you do not create object code at all. When you assemble code at $1000, for example, you cannot normally set the program counter back to $0F00 to assemble code there. This is allowed for label definition, but you must then return to an address which was higher than the address into which the last byte of object code was assembled.

4. .BYTE expression

The .BYTE pseudo-op is used to place one-byte values into the object code at the location contained in the program counter. Any legal ASSEMBLER 64 expressions, separated by commas, may be used as operands. The number is limited only by line length and the length of the ASSEMBLER 64's buffer. Any expressions may be used, but the expression must evaluate to a one-byte value, or an "ILLEGAL QUANTITY ERROR" occurs. Two-byte values can be modified with ">" and "<" in order to take the high or low byte, respectively. A one-byte value lies in the range 0 to 255 or $FF80 to $FFFF. The higher values are allowed because they normally signify negative numbers from −1 to −128. Therefore the line ".BYTE −1" is

allowed. .BYTE can be used to define tables such as jump tables or pointers. You can also "hide" commands, such as the BIT command:

```
        .BYTE $2C ; ABSOLUTE BIT INSTRUCTION
LABEL1 LDA #-1 ; HIDDEN LDA INSTRUCTION
```

5. .WORD expression

The .WORD pseudo-op is used in order to place two-byte addresses into the object code at the location contained in the program counter. For example the following statements:

```
START = $C000
      .WORD START
```

Would assemble the bytes 00 0C (the value of the symbol START, least significant byte first) into the object code. The address is stored with the least significant byte first followed by the most significant byte.

.WORD address

is equivalent to the statements:

.BYTE <address;>address

The .WORD pseudo-op and the .BYTE pseudo-op permit multiple values on a line, separated by semicolons. The .WORD pseudo-op is most often used for creating address tables.

6. .FILE device number,"filename"

The .FILE pseudo-op is used to chain several source programs. The syntax is as follows:

.FILE device number, "filename"

where device number is 8 for the disk drive or 1 for the datasette, and "filename" is the name of the assembly language source program which is to be loaded next. If you are writing a very long assembly language program, you can break it up into several parts and chain these together with .FILE. The last file in this chain must contain an .END pseudo-op that specifes the first file of the chain.

7. .IF expression

The .IF pseudo-op is used for conditional assembly. The syntax is as follows:

IF expression : .GOTO line-number

The argument expression is evaluated in both pass 1 and pass 2. If the expression is not zero, the code following the .IF in the same line is performed. Usually, this will be a .GOTO to direct the assembly to a different line. The additional code in the line must be separated by colons.

With .IF, .GOTO, and symbol redefinitions, it is possible to create assembler loops. Although .IF only tests for zero, other comparisons are possible by using simple techniques. For example, shifting 15 bits to the right yields a result of 1 if the expression was negative, and 0 if positive. Two numbers may be compared by subtracting one from the other and testing the result for positive or negative.

8. .GOTO line-number

The .GOTO pseudo-op instructs the assembler to continue assembly at the line number given as the argument.

.GOTO line-number

This line number may also be an expression. The line number must be contained in the currently loaded program (if you are using .FILE to chain multiple source programs). You cannot jump between different files. This line number may be located either before or after the line number containing the .GOTO pseudo-op. When used with .IF and redefining symbols, it's possible to build a loop for conditional assembly. Try the following example:

```
10 SYS 32768 ; CALL THE ASSEMBLER
20 .OPT P ; LISTING TO SCREEN
30 OFFSET <- 5 ; NUMBER OF LOOPS
40 LDA $C000 + OFFSET
50 OFFSET <- OFFSET - 1 ; DECREMENT
60 .IF OFFSET : .GOTO 40
70 .END
```

9. .GTB

This pseudo-op stands for Go To BASIC. It has no argument and simply returns control to BASIC. The BASIC commands in following program lines will be executed. You may return to the assembler by using SYS40954.

You should note that the BASIC commands that can be executed before return to assembler are limited. Some BASIC statements may overwrite the work areas used by ASSEMBLER 64 and should not be executed. In particular, the INPUT command, or any other basic commands which

writes to byte 9 of the BASIC input buffer (address $0209) must be avoided. The GET statement is allowed. You should never return control to the user during assembly.

10. .ASC "text"

This pseudo-op places the ASCII value(s) for the "text" into the object code at the location contained in the program counter. The text is enclosed in quotation marks. It is thereby possible to insert cursor or color control characters into the text. The text can be up to 55 characters long. Longer texts must be divided up into several .ASC statements. The MOS standard uses the .BYTE pseudo-op for this purpose, in which strings are enclosed in apostrophes. You should take this into account when converting programs. Note the use of the double quotes instead of single quotes.

11. .SYS expression

This pseudo-op allows machine language programs to be called during assembly. The value of expression determines the jump address. This pseudo-op is identical to the SYS command in BASIC. The routine located at the address specified by expression is called is called during both pass 1 and pass 2. The SYS command can be used by those familiar with the internal workings of ASSEMBLER 64 to create custom pseudo-ops.

12. .STM expression

This pseudo-op is used to raise the lower boundary of the symbol table. The symbol table grows downward from the end of the storage ($8000), exactly as strings are saved in BASIC. At the start of assembly, this lower boundary is set to the end of the BASIC program and variables. You can set it higher if you are working with .FILE or buffered object code (.OPT O). If the space for the symbol table is too small, the message "SYM TABLE OVERFLOW" is given and the assembly stopped.

13. .SST device number, secondary address, "filename"

Symbol tables may be saved to storage devices such as the floppy disk, and from there loaded in again. .SST is executed in pass 1 only, and saves the symbol table that has been generated up to that point.

The first argument is the device number, normally 8 for the disk drive. The secondary address can lie between 2 and 14. The filename is given as in an OPEN command, and therefore requires an ",S,W" following the name (for sequential and write).

This pseudo-op is required if you want to later print a sorted list of symbols and labels. The program SYMPRINT then uses this file to list the symbols to your printer.

The .SST command is also useful when assembling source programs separately, but which must access subroutines from the other programs. Simply save the symbol table at the end of first assembler program and read this same symbol table into the second program using .LST.

14. .LST device number, secondary address, "filename"

This pseudo-op loads the symbol table that was saved by the .SST pseudo-op. You can use .LST to load the a symbol table created by other programs, such as a table of kernal routines. Duplicate symbols are not checked. The last definition of a duplicate symbol is used and previous definitions are simply ignored. Overflow of the symbol table is not recognized when loading, although an error will occur as soon as you try to define another symbol.

15. .FLP expression

If you often use the floating-point arithmetic of the BASIC interpreter, you can use .FLP to place floating-point constants into the object code. This simplifies the use of floating-point routines. One or more floating-point constants separated by commas can follow the .FLP command, for example:

```
.FLP 10, 1E8
```

Each floating-point number occupies 5 bytes; therefore our example generates 10 bytes. Note that only the first three bytes of the converted number are printed in the object code listing.

16. .END [device, "filename"]

This pseudo-op ends a source program and is optional. .END executes a .GTB at the end of pass 2. If there are additional BASIC statements following the .END pseudo-op, they will be executed.

You can, for example, call the machine language program just assembled with a SYS-statement.

When chaining source programs, .END must have the additional arguments. The arguments are in the same format as the .FILE pseudo-op and direct the assembler to re-load the first source program at the end of pass 1 and continue with pass 2 at the line containing the SYS 32768. "filename" must therefore be the name of the first program in the chain (which contains the SYS 32768). "filename" has no further effect in pass 2.

17. .SYM

This pseudo-op can be used to list a table of all the defined symbols and their values after the assembly of the program. This list is sent to the screen or other device according to the output option (.OPT P). Four symbols, together with their values in hexadecimal form, are printed per line. If you want a different number of symbols per line, you can use this number as an argument for the .SYM command. .SYM is useful when working on the screen, for example. The symbols are listed in the reverse order from that in which they were defined. If you want an alphabetically sorted list, you must save the symbol table with .SST and use the program SYMPRINT found on your ASSEMBLER 64 distribution disk.

18. .PAGE page-length,left-margin offset

This pseudo-op has three different functions and serves to control the assembly language listing. Without additional parameters, it forces a form feed in the listing. This allows you to place a certain section of an assembler listing on a new page. ASSEMBLER 64 automatically inserts a form feed after every 60th line, and begins the next page with a title and the current page number. If you want to change the page length, you can set the number of lines per page with the .PAGE command, for example:

```
.PAGE 66
```

This instructs ASSEMBLER 64 to write 66 lines on a page. Values up to 255 are accepted. An additional function is the determination the left margin. This is useful for printed listings which you want to put in a notebook. The second parameter of .PAGE gives the number of spaces to be printer in front of each assembler line in the listing. The standard value is zero. With

```
.PAGE ,10
```

the listing can be indented 10 characters. The comma is necessary in order to denote the 10 as the second parameter. The two parameters can also be combined:

```
.PAGE 66,10
```

19. .TITLE "text"

This allows you to add text to the standard title

```
ASSEMBLER 64 V2.0 PAGE 1
```

which appears on every page of the listing. This text is given after the .TITLE command within quotation marks, such as:

```
.TITLE "HARDCOPY ROUTINE"
```

This text will then be placed before the standard title, and we get:

HARDCOPY ROUTINE ASSEMBLER 64 V2.0 PAGE 1

20. .OPT options{,options}

The .OPT pseudo-op stands for OPTion and gives you control over the assembly listing and the object code. This syntax is the following:

.OPT option,option,option ...

The following options are available:

P – Print. You select this option when you want the assembly listing to appear on the screen. All other P options (see below) also output to the screen because the screen is the fastest output medium. The listing will be formatted automatically. Lines which contain errors or a .FILE command will be printed in passes 1 and 2 regardless of the P option.

P# – Print to file. With this option, you can send a listing to the printer, for example. In order to do so, you must first open a logical file before the SYS 32768 with an OPEN command, such as OPEN 1,4. The logical file number (1 in our example) then replaces the number sign (#), such as .OPT P1. using this technique, you can also write the assembly listing to disk or cassette with the appropriate OPEN command. You can specify that a line feed (CHR$(10)) be sent after each carriage return (CHR$(13)) when selecting the logical file number in BASIC. This accomplished by using a logical file number greater than 127, such as OPEN 130,4 and then .OPT P130.

P = expression – With this option you can direct the output to a routine of your own. The start address of your routine must be given as the expression. The character to be outputted is passed in the accum ulator. A zero indicates the last character (close file). This allows custom output devices to be used (such as an interface on the user port).

O – Object means object code output. Without additional characters, the object code goes to a special buffer directly above the assembler program, where array variables normally lie; the same pointers are also used.

OO – Object at origin. This option writes the object code directly to the memory locations for which it was written. This is vary useful for quickly testing programs, and allows maximum freedom when moving the program pointer. Saving code to tape is also made possible using the monitor. If an assembly language program is intended to run in the memory range where the source program or assembler lies, this method may naturally not be used.

O# – As with P#, this allows output of the object code to a file. The file must be previously opened as a program file for writing (secondary address 1), such as OPEN 1,8,1"PROGRAM". With .OPT O1, the object code goes to this file. First ASSEMBLER 64 writes the start address to the file, and then the generated code. If the assembler operation ends normally, the program file will be closed again. The machine language program created in this manner can be loaded directly with LOAD or with a monitor. Note that .OPT O# to a cassette is not possible. See the next option and the appendix.

O = expression – This allows the object code to be sent to a user-defined routine with the same syntax as the .OPT P= command. The object code output routine must be somewhat more complicated because it is called only once per assembler line. Some symbols which are required are found in the appendix. The most important is LENGTH, which gives the number of bytes to be output minus 1. If length is zero, for example, one byte must be output. You routine must be test for two special values. A value of $C0 means "close the file." Otherwise, LENGTH contains a small number from zero on up. The data to be output are stored in two places. The first three bytes are stored in the zero page at address OP. If more than three bytes of object code are created (for .BYTE, .WORD, .ASC, for example), the additional bytes are stored at address OBJBUF. Your output routine may change any registers or flags (with the exception of the decimal flag). Caution is advised in using the zero page however. A program is listed in the appendix which makes it possible to output the object code to a file in hex format. It is therefore possible in principle to save data directly to the datasette.

M – If you work with macros, you can decide whether you want the entire macro containing the actual parameters to be listed for each macro call, or just the line containing the macro call. If you do not enter this command, the complete macro will be listed. You can suppress this with .OPT M and cause only the line with the macro call to be listed.

N – You can cancel the output options at any time with .OPT N. N cancels all of the options except the M option. If an option is supposed to remain in effect or switched on again later, add that option. If, for example, you want to turn off the screen listing, but still want the object code to go to file 2, you would write

```
.OPT N,O2
```

and

```
.OPT P
```

when the listing is to go to the screen again.

D. A SAMPLE PROGRAM

The following example program writes the contents of the zero page at line LINE on the screen. It illustrates the general use of the assembler.

```
10 SYS 32768 ; CALL ASSEMBLER
20 .OPT P,00
30 *= $C000  ; PGRM START ADDR
40 LINE = 10 ; LINE 10 ON SCRN
50 SCRMEM = $400 ; SCRN MEMORY
60 CLRMEM = $D800 ; COLOR MEMORY
70 COLOR = 1 ; COLOR IS WHITE
80 LDX #0 ; ZERO INDEX REG
90 LOOP LDA 0,X ; GET BYTE
100 STA SCRMEM+(40*LINE),X ; PUT IN SCRN MEMORY
110 LDA #COLOR
120 STA CLRMEM+(40*LINE),X ; SET COLOR
130 INX ; NEXT BYTE
140 BNE LOOP
150 RTS ;DONE
160 .END
```

If you start assembler this source program by typing RUN, the following listing the screen:

```
2
ASSEMBLER 64 V2.0 PAGE 1

20:   C000                    .OPT P,00
30:   C000                    *=    $C000 ; PGRM START ADDR
40:   000A            LINE    =     10    ; LINE 10 ON SCRN
50:   0400            SCRMEM  =     $0400 ; SCRN MEMORY
60:   D800            CLRMEM  =     $D800 ; COLOR MEMORY
70:   0001            COLOR   =     1     ; COLOR IS WHITE
80:   C000 A2 00              LDX   $0    ; ZERO INDEX REG
EX REGISTER
90:   C002 B5 00      LOOP    LDA   0,X   ; GET BYTE
100:  C004 9D 90 05           STA  SCRMEM+(40*LINE),X  ; PUT
IN SCRN MEMORY
110:  C007 A9 01              LDA  #COLOR
120:  C009 9D 90 D9           STA  CLRMEM+(40*LINE),X ; SET
COLOR
130:  C00C E8                 INX         ; NEXT BYTE
140:  C00D D0 F3              BNE  LOOP
150:  C00F 60                 RTS         ; DONE
]C000-C010
NO ERRORS
```

In the following example, the object code is sent directly to disk and the listing is sent to the printer. The source program consists of several individual programs.

```
10 OPEN 1,8,1, "0:OBJECT CODE"
20 OPEN 2,4 : REM PRINTER
30 SYS 32768
40 .OPT O1,P2
50 ; ASSEMBLER COMMANDS
...
1000 .FILE 8, "PROGRAM 2"
```

PROGRAM 2 contains

```
10 ; ADDITIONAL COMMANDS
...
1000 .FILE 8, "PROGRAM 3"
```

PROGRAM 3 contains

```
10 ; ADDITIONAL COMMANDS
...
1000 .END 8, "PROGRAM 1"
```

whereby PROGRAM 1 is the program which contains the SYS 32768.

E. MACROS

We now come now to a powerful feature of ASSEMBLER 64 – MACROS. What are macros and what are they used for?

With macros we have the ability to combine a series of instructions and assembler directives and give them a name. If you have defined a macro in this manner, you can later insert this set of instructions into the source code as often as desired by simply using the name of the macro. An example will make this clear.

In machine language programs, one repetitive task often comes up in programming – namely incrementing the contents of a 16-bit variable located in consecutive zero page locations. The instructions to do this might look like this:

```
    INC POINTER
    BNE LABEL
    INC POINTER+1
LABEL ...
```

At another place you might have to increment a different variable called TEMP:

```
    INC TEMP
    BNE LABEL1
    INC TEMP+1
LABEL1 ...
```

With macros we can define a set of instructions once and use this definition later. To define a macro, two new pseudo-ops are used.

The first declares the macro definition, and the second ends it. In order to able to refer to a macro later, it must have a name. The same conventions apply as for other symbols (first character must be a letter, then letters, digits, or periods, eight significant places). Our definition looks like this:

```
INC.PNT .MAC ADDRESS
        INC ADDRESS
        BNE .LABEL
        INC ADDRESS+1
LABEL   .MEND
```

The name of this macro is INC.PNT. A macro definition is introduced with the pseudo-op .MAC. Parameters may follow. Here we have a parameter called ADDRESS. Next the executable instructions follow in their standard form. One special feature is found in the line BNE .LABEL. The last line

contains the label definition and the end of the macro definition with .MEND. Now we can call the newly-defined macro:

```
'INC.PNT POINTER
```

This line replaces the above set of instructions. We write an apostrophe followed by the macro name and any parameters. In our case there was one parameter, although a macro can have no parameters, or several parameters separated by commas. When assembled, the macro is replaced by the instructions:

```
        INC POINTER
        BNE LABEL:00
        INC POINTER+1
LABEL:00
```

The next example illustrates a macro without parameters.

```
RAM     .MAC
        SEI
        LDA $01
        AND #%11111110
        STA $01
        .MEND
```

This macro requires no parameters and no so-called local labels – labels within the macro definition. Macros without parameters generate the same code each time and can in principle be replaced by subroutines. Macros are aids during the assembly and create object code each time it is used. Subroutines can be thought of as aids during run-time, and are found only once in the object program.

Macros are especially useful in combination with conditional assembly. If you have macros ready for a variety fundamental tasks, the main program can consist of a set macro calls.

A few notes about using macros:

Macros must be defined at the start of the assembly language source, before they are called. If you are chaining source programs using .FILE, all macros must all be contained in the first program. If you define labels within a macro, a period must be placed before references to the label, as illustrated earlier. This also applies within expressions. Such labels are only significant to six characters. If you call such macros several times and output the symbol table, the labels are listed as many times, together with different values. In order to distinguish these from each other, the name is followed by a colon and the number of the label, for example:

```
LABEL:00   0006   LABEL:01   C020   LABEL:02   C035
```

The number zero indicates the label value within the definition, relative to the start of the macro.

If labels are defined with a macro, different names must be used within different macros, or a "REDEFINITION ERROR" will occur. Parameters may have the same names because these are replaced by the actual values during a macro call anyway. Arbitrary ASSEMBLER 64 expressions can be used in a macro call; these are calculated by the assembler and transmitted as parameters, for example:

```
'INC.PNT POINTER-8*2
```

Here, for example, the value of pointer is taken and the result of 8 times 2 is subtracted from it. The order of evaluation can be determined through the use of parentheses as usual.

As an example, we have a program which consists almost entirely of macro calls. Two macros are defined. The first serves to set the cursor. The operating system of the Commodore 64 places this routine at our disposal. The macro with the name CURSOR expects two parameters. The first is the line in which the cursor is to be placed, and the second is the column. If we want to set the cursor at a specific place in our program, we need only call the macro, for example:

```
'CURSOR 10,20
```

The second macro serves to output text. The parameter is the address of the text. The string must be terminated by a zero byte.

In the program you find first the definition of the two macros and then the actual program which consists only of four macro calls and an RTS. The strings are listed at the end of the program.

The source program is listed on the next page followed by the assembly listing:

```
50 OPEN128,4,5
100 SYS 32768
110 .OPT P128,00
120 ; DEMO PROGRAM FOR MACROS
130 ;
140 ; SET CURSOR
150 CURSOR .MAC LINE, COL
160 LDX #COL
170 LDY #LINE
180 STX $D6
190 STY $D3
200 JSR SETCRSR ; SET CURSOR
210 .MEN
220 ;
230 ; STRING OUTPUT
240 PRTSTR .MAC TEXT
250 LDA #<TEXT
260 LDY #>TEXT
```

```
270 JSR STROUT ;O/P TEXT TO SCREEN
280 .MEN
290 ;
300 SETCRSR = $E56C
310 STROUT = $AB1E
320 ;
330 *= $C000
340 ;
350 'CURSOR 10,10
360 'PRTSTR TEXT1
370 'CURSOR 0,20
380 'PRTSTR TEXT2
390 RTS
400 ;
410 TEXT1 .ASC "TEXT LINE # 1": .BYT 00
420 TEXT2 .ASC "TEXT LINE # 2": .BYT 00
430 ;
440 .END
```

Here's the assembly listing:

```
ASSEMBLER-64  V2.0     PAGE 1

110:    C000                    .OPT P128,00
120:                 ; DEMO PROGRAM FOR MACROS
130:                 ;
140:                 ; SET CURSOR
150:    .            CURSOR     .MAC LINE,COL
160:    .                       LDX #COL
170:    .                       LDY #LINE
180:    .                       STX #D6
190:    .                       STY #D3
200:    .                       JSR SETCRSR  ;SET CURSOR
210:    .                       .MEN
220:                 ;
230:                 ;  STRING OUTPUT
240:    .            PRTSTR     .MAC TEXT
250:    .                       LDA  #<TEXT
260:    .                       LDY  #>TEXT
270:    .                       JSR  STROUT  ;O/P TEXT TO SCREEN
280:    .                       .MEN
290:                 ;
300:    E56C         SETCRSR    =    $E56C
310:    AB1E         STROUT     =    $AB1E
320:                 ;
330:    C000                    *=   $C000
340:                 ;
350:    C000                    'CURSOR 10,10
+       C000 A2 0A              LDX  #COL
+       C002 A0 0A              LDY  #LINE
+       C004 86 D6              STX  $D6
```

```
+       C006 84 D3          STY  $D3
+       C008 20 6C E5       JSR  SETCRSR  ;SET CURSOR
+       C00B                .MEN
360:    C00B                'PRTSTR TEXT1
+       C00B A9 25          LDA  #<TEXT
+       C00D A0 C0          LDY  #>TEXT
+       C00F 20 1E AB       JSR STROUT  ;O/P TEXT TO SCREEN
+       C012                .MEN
370:    C012                'CURSOR 0,20
+       C012 A2 14          LDX #COL
+       C014 A0 00          LDY #LINE
+       C016 86 D6          STX $D6
+       C018 84 D3          STY $D3
+       C01A 20 6C E5       JSR SETCRSR ;SET CURSOR
+       C01D                .MEN
380:    C01D                'PRTSTR TEXT2
+       C01D A9 33          LDA #<TEXT
+       C01F A0 C0          LDY #>TEXT
+       C021 20 1E AB       JSR STROUT  ;O/P TEXT TO SCREEN
+       C024                .MEN
390:    C024 60             RTS
400:                ;
410:    C025 54 45 58 TEXT1 .ASC "TEXT LINE # 1"
410:    C032 00             .BYT 00
420:    C033 54 45 58 TEXT2 .ASC "TEXT LINE # 2"
420:    C040 00             .BYT 00
430:                ;
```

Let's take a closer look at the listing. You recognize that within the macro definition, an apostrophe appears instead of the program counter. The object code field is empty because no code is created by the macro definition (lines 150-210, 240-280).

The first macro call is in line 350. The actual program counter as well as the code created appear in the listing. A plus sign (+) appears in place of the line number, which shows that the created code comes from a macro call. You recognize that the symbols LINE and COLUMN have the values which they were assigned by the macro call. The subsequent macro calls proceed in the same manner.

If you have many macros in your source program or you call certain macros often, you have the option of supressing the macro-created code in the assembly listing. Only the line containing the actual call will appear. The option .OPT M performs this function. See the next example:

```
ASSEMBLER-64  V2.0    PAGE 1

110:    C000                .OPT P128,M,00
120:              ; DEMO PROGRAM FOR MACROS
130:              ;
```

```
140:                ; SET CURSOR
150:    .           CURSOR    .MAC LINE,COL
160:    .                     LDX #COL
170:    .                     LDY #LINE
180:    .                     STX #D6
190:    .                     STY #D3
200:    .                     JSR SETCRSR  ;SET CURSOR
210:    .                     .MEN
220:                ;
230:                ;  STRING OUTPUT
240:    .           PRTSTR    .MAC TEXT
250:    .                     LDA  #<TEXT
260:    .                     LDY  #>TEXT
270:    .                     JSR  STROUT  ;O/P TEXT TO SCREEN
280:    .                     .MEN
290:                ;
300:    E56C        SETCRSR   =    $E56C
310:    AB1E        STROUT    =    $AB1E
320:                   ;
330:    C000                  *=   $C000
340:                   ;
350:    C000                  'CURSOR 10,10
360:    C00B                  'PRTSTR TEXT1
370:    C012                  'CURSOR 0,20
380:    C01D                  'PRTSTR TEXT2
390:    C024 60               RTS
400:                   ;
410:    C025 54 45 58 TEXT1   .ASC "TEXT LINE # 1"
410:    C032 00               .BYT 00
420:    C033 54 45 58 TEXT2   .ASC "TEXT LINE # 2"
420:    C040 00               .BYT 00
430:                   ;
```

Supressing the macros makes the listings shorter and often easier to read. In the next example we have added a .SYM pseudo-op as statement number 115. This pseudo-op prints a list of the symbols and their values together with the defined macros and the number of times which they were defined as a two-digit hexadecimal number. The first part of the listing is the same as the previous page. Only the symbol table and macro table are printed as follows:

```
ASSEMBLER -64   V2.0      PAGE 2

SYMBOLTABLE:
TEXT2      C033    TEXT1     C025    TEXT      C033  COL   0014
LINE       0000    STROUT    AB1E    SETCRSR   E56C
7 SYMBOLS DEFINED

MACROTABLE:
PRTSTR     02      CURSOR    02
2 MACROS DEFINED
```

F. ERROR MESSAGES

ASSEMBLER 64 has a set of error messages. Errors are printed in both pass 1 and pass 2. If the assembler recognizes an error, 4 asterisks followed by the error message is displayed. The line containing the error will then be displayed on the screen, regardless of the .OPT P settings. For a syntax error, a digit will also be displayed in front of the four asterisks which describes the error in greater detail. There are 10 different types of syntax errors which can occur. They are listed below. Still other errors can occur when using macros; these are indicated by a prefixed letter.

Some errors are "fatal," meaning that they cause the assembly to stop. An exclamation point is displayed in front of lines containing fatal errors. The assembly is stopped after the message is displayed. The first byte of the object code created for such a line is a zero, which is the 6502 BRK command. If you try to execute such a program, a BRK command is executed when it comes to the erroneous line, which either performs a warm start, or returns you the monitor, if it is loaded. In general, you should first correct the errors before you execute an assembly language program.

One type of error which ASSEMBLER 64 cannot detect is a phase error. This error does not usually occur, but can be encountered with certain combinations of conditional assembly containing .BYTE or .WORD pseudo-ops. A phase error occurs when the program counter is different in pass 2 than it was in pass 1. You can recognize a phase error with an .IF instruction:

```
PHASE .IF PHASE-* : PHASE ERROR
```

Normally, PHASE has the same value as the program counter and the code behind the colon is never executed. If a phase shift occurs, the result is not zero and the additional statement results in a syntax error which you can recognize.

Error statistics

Before the start of pass 2, ASSEMBLER 64 outputs the number of errors in pass 1, if any were found. For example:

```
2 ERRORS IN PASS 1
```

After pass 2, when the assembly is complete, the number of errors in pass 2 is displayed. If the assembly was error-free, the message

```
NO ERRORS
```

is displayed. If errors were encountered, that number is displayed. For example:

```
4 ERRORS
```

Messages

SYNTAX
– This error message is preceded by a digit which describes the error in greater detail. These digits have the following meaning:

0 – Label for empty assignment not allowed (the line contains only one string).
1 – Illegal opcode
2 – Illegal addressing mode – this command may not be used with this addressing mode.
3 – Unknown operator in expression (unallowed character in an expression)
4 – Unpaired parentheses
5 – Illegal expression – illegal character in an expression, or an empty string "".
6 – Missing comma – a pseudo-op is expecting a comma
7 – Illegal pseudo-op. The .XXX string is not recognized as a pseudo-op.
8 – Symbol does not start with a letter. A symbol was expected, but an alphabetic character was not found.
9 – Opcode with unallowed addressing mode.

The following syntax errors can occur for macros:

B – .MEND command without previous .MAC
C – Unclosed macro definition
D – Nested macro definition – macros within macros are not allowed.
F – Illegal number of parameters. The number of parameters in the macro call does not match the number in the macro definition.

ILLEGAL QUANTITY
– The expression evaluated to a value which lies outside the borders for this command or pseudo-op. The expression yields a value greater than 65535.

OVERFLOW
– The input buffer which ASSEMBLER 64 uses in order to decode source lines is to small. Divide the line into several instructions or use a temporary variable in order to simplify the expression.

BRANCH OUT OF RANGE
– A relative jump (branch command) over a distance greater than 128 bytes was attempted.

REDEFINITION
– An attempt was made to define a symbol twice without using the redefinition operator.

UNDEF'D STATEMENT
– A label or expression is not defined.

REVERSAL
– An attempt was made to assemble code at an address which is lower than the last address. This error does not occur when you assemble directly to memory. This is a fatal error, as are all of the following.

SYM TABLE OVERFLOW
– You have tried to define more symbols than space in the symbol table permits. Either set the minimum lower with .STM, or divide your program into several parts. This error message can also appear when loading a source program with .FILE if the program is too large and part of the symbol table has been overwritten. Divide the program into smaller parts.

OUT OF MEMORY
– The buffer for the object code (.OPT O mode) is too small. You should choose some other type of output, such as disk.

UNDEF'D STATEMENT
– A GOTO to a non-existent line (exactly as in BASIC). In contrast to the error named before, this one is fatal.

DEVICE NOT PRESENT
– The addressed device is not present on the bus, or does not answer.

IEEE
– Another error on the IEEE bus.

DISK
– Disk error. The disk drive error message was given just prior to this.

G. Appendix

The following source program is another example of the use of ASSEMBLER 64. It demonstrates outputting of object code by a user-defined routine. It sends each byte in hex format to a previously opened file with the logical file number 1. It is therefore possible to write the object code directly to the datasette, for example. It is possible to read code in this format with the BASIC program following it.

```
100 SYS 32768 ;CALL ASSEMBLER
110 .OPT
120 LENGTH = $4E ;BYTES TO O/P - 1
130 OP = $4B ;BUFFER FOR FIRST 3 BYTES
140 ADDR = $56 ;PGRM START ADDR
150 OBJBUF = $15B ;BUFFER FOR ADDITIONAL BYTES
160 CHKOUT = $FFC9 ;O/P TO LOGICAL FILE
170 CLRCH = $FFCC ;O/P TO DEFAULT
180 PRINT = $FFD2 ; O/P A CHARACTER
190 CLOSE = $FFC3
200 LF = 1 ;LOGICAL FILENUMBER
210 *= $C000 ;START ADDRESS
220 LDA LENGTH
230 CMP #$C0 ;CLOSE
240 BEQ CLOSEF
250 LDX #LF : JSR CHKOUT ;OUTPUT TO LOGICAL FILE 1
260 LDX #0 : LDA LENGTH
270 CMP #$80 ;OPEN
280 BEQ STARTADDR
290 OUT LDA OP,X
300 OUT1 JSR WROB ;OUTPUT BYTE AS HEXNUMBER
310 CPX LENGTH
320 BEQ EX1
330 INX
340  CPX #3
350 BCC OUT
360 LDA OBJBUF-3,X
370 JMP OUT1
380 EX1 JMP CLRCH
390 CLOSEF LDA #LF
400 JMP CLOSE
410 STARTADDR LDA ADDR : JSR WROB ;START ADDR LOW
420 LDA ADDR+1 : JSR WROB ;START ADDR HIGH
430 JMP CLRCH
440 WROB PHA ;O/P BYTE AS HEX NUM
450 LSR : LSR : LSR :LSR ;UPPER NYBBLE
460 JSR ASCII
470 PLA
```

```
480 AND #%1111 ;LOWER NYBBLE
490 ASCII CLC
500 ADC #-10
510 BCC ASC1
520 ADC #6
530 ASC1 ADC #"9"+1
540 JMP PRINT
550 .END
```

If you assemble this program, you get the following assembly listing:

```
ASSEMBLER-64 V2.0     PAGE 1

110: C000                    .OPT P128,00
120: 004E          LENGTH    =    $4E   ;BYTES TO THE O/P - 1
130: 004B          OP        =    $4B   ;BUFFER FOR FIRST 3 BYTES
140: 0056          ADDR      =    $56   ;PGRM START ADDR
150: 015B          OBJBUF    =    $15B  ;BUFFER FOR MORE BYTES
160: FFC9          CHKOUT    =    $FFC9 ;O/P TO LOGICAL FILE
170: FFCC          CLRCH     =    $FFCC ;O/P TO DEFAULT
180: FFD2          PRINT     =    $FFD2 ;O/P A CHARACTER
190: FFC3          CLOSE     =    $FFC3
200: 0001          LF        =    1     ;LOGICAL FILENUMBER
210: C000                    *=   $C000 ;START ADDRESS
220: C000 A5 4E              LDA  LENGTH
230: C002 C9 C0              CMP  #$C0  ;CLOSE
240: C004 F0 24              BEQ  CLOSEF
250: C006 A2 01              LDX  #LF
250: C008 20 C9 FF           JSR  CHKOUT ;O/P TO LOGICAL FILE 1
260: C00B A2 00              LDX  #0
260: C00D A5 4E              LDA  LENGTH
270: C00F C9 80              CMP  #$80  ;OPEN
280: C011 F0 1C              BEQ  STARTADDR
290: C013 B5 4B    OUT       LDA  OP,X
300: C015 20 3C C0 OUT1      JSR  WROB  ;O/P BYTE AS HEX #
310: C018 E4 4E              CPX  LENGTH
320: C01A F0 0B              BEQ  EX1
330: C01C E8                 INX
340: C01D E0 03              CPX  #3
350: C01F 90 F2              BCC  OUT
360: C021 BD 58 01           LDA  OBJBUF-3,X
370: C024 4C 15 C0           JMP  OUT1
380: C027 4C CC FF EX1       JMP  CLRCH
390: C02A A9 01   CLOSEF     LDA  #LF
400: C02C 4C C3 FF           JMP  CLOSE
410: C02F A5 56 STARTADDR    LDA  ADDR
410: C031 20 3C C0           JSR  WROB   ;START ADDR LOW
420: C034 A5 57              LDA  ADDR+1
420: C036 20 3C C0           JSR  WROB   ;START ADDR HIGH
430: C039 4C CC FF           JMP  CLRCH
```

```
440: C03C 48      WROB    PHA          ;OUTPUT BYTE AS HEX #
450: C03D 4A              LSR
450: C03E 4A              LSR
450: C03F 4A              LSR
450: C040 4A              LSR          ;UPPER NYBBLE
460: C041 20 47 C0        JSR  ASCII
470: C044 68              PLA
480: C045 29 0F           AND  #%1111 ;LOWER NYBBLE
490: C047 18      ASCII   CLC
500: C048 69 F6           ADC  #-10
510: C04A 90 02           BCC  ASC1
520: C04C 69 06           ADC  #6
530: C04E 69 3A    ASC1   ADC  #"9"+1
540: C050 4C D2 FF        JMP  PRINT
]C000-C053
NO ERRORS
```

If you assemble this program, you can write the object code in hex format to the datasette with this format:

```
100 OPEN 1,1,1,"OBJECT CODE" : REM WRITE TO TAPE
110 SYS 32768
120 .OPT P,O=$C000 ; OBJECT CODE TO CUSTOM ROUTINE
```

The program can be loaded from tape with a small loader program in BASIC.

```
100 OPEN 1,1,0,"OBJECT CODE" : REM READ FROM TAPE
110 GOSUB 1000 : AD = A : REM LOW BYTE OF START ADDRESS
120 GOSUB 1000 : REM HIGH BYTE OF START ADDRESS
130 AD = A*256 + AD : REM START ADDRESS
140 IF ST=64 THEN CLOSE 1 : END : REM PROGRAM END
150 GOSUB 1000 : REM READ BYTE
160 POKE AD,A : AD = AD + 1
170 GOTO 140
1000 REM READ HEX NUMBER
1010 GET#1, A$,B$
1020 H = ASC(A$)-48+(A$>="A")*7 : REM HIGH NYBBLE
1030 L = ASC(B$)-48+(B$>="A")*7 : REM LOW NYBBLE
1040 A = L+16*H : RETURN
```

Your ASSEMBLER 64 distribution diskette contains a BASIC program called "SYMPRINT". This program serves to output a symbol table in alphabetic order, which you have written to disk previously with .SST.

The program asks for the name of the symbol table on disk as well as the number of output device (3=screen, 4=printer, 8=disk). For disk output, you must give the name of the file to which the symbol table will be written. Finally, you can determine how many symbols will be printed per line. Two fit per line on the screen, 4 on a printer. The output format corresponds to that of the .SYM command when assembling.

THE MONITOR

MONITOR 64 is an extended machine language monitor that has features not found in more conventional software. It can be loaded concurrently with ASSEMBLER-64 and thus forms a complete machine language development package.

A. Summary of MONITOR 64 Commands

Here is a list of the commands that can be performed with MONITOR-64:

Commands:

R	Register display	display register contents
M	Memory display	display memory contents
G	Go	execute machine language program
L	Load	load machine language program
S	Save	save machine language program
D	Disassemble	disassemble machine language prog.
C	Compare	compare memory areas
T	Transfer	move memory area
H	Hunt	search through memory range
F	Fill	fill memory range with value
B	Bank	select memory configuration
W	Walk	single-step mode
Q	Quicktrace	trace with break points
U	Breakpoint	set breakpoint
X	Exit	return to BASIC

B. LOADING MONITOR-64

The monitor occupies 3K bytes of memory from $C000 to $CBFF outside the BASIC area and is loaded from diskette. Type:

LOAD "MONITOR 64",8,1

and press <RETURN>. The messages

SEARCHING FOR MONITOR 64
LOADING

MONITOR 64 V2.0 IS LOADING ...

appear on the screen. Once loaded, the monitor responds with

*** MONITOR 64 V2.0 ***
(C) 1984 DATA BECKER GMBH

C*

and displays the register contents.

All monitor input and output is done using 2 or 4 digit hexadecimal numbers.

C. COMMAND Descriptions

Here is a description of the MONITOR-64 commands:

1. Switch memory configuration >BX

With this command you can have access to the entire memory of the Commodore 64. After starting the monitor, all commands operate on the normal memory configuration. With >BA you can switch the memory configuration to all RAM, while >BC also adds the character generator. You can switch back to the normal ROM configuration with >BR. This configuration effects only the commands

M, D, C, T, H, and F

The following table illustrates the three configurations.

Address range	>BR	>BA	>BC
$E000 – $FFFF	ROM	RAM	RAM
$D000 – $DFFF	I/O	RAM	CHAR ROM
$C000 – $CFFF	RAM	RAM	RAM
$A000 – $BFFF	ROM	RAM	RAM
$0000 – $9FFF	RAM	RAM	RAM

2. Compare memory areas >C XXXX YYYY ZZZZ

The memory area from addresses XXXX through YYYY is compared with the area starting at ZZZZ byte by byte. Any address whose contents differ are displayed.

Example: >C 8000 8100 9000
8056

The contents of address $8056 differ from the contents of address $9056.

3. Disassemble a machine language program >D XXXX YYYY

The machine language program beginning at address XXXX through YYYY will be displayed in mnemonic (operation code) form. If the ending address

YYYY is omitted, only one line is displayed. Three question marks will be displayed for invalid instructions.

```
Example: >D B016 B021
         >.  B016   20 90 AD  JSR $AD90
         >.  B019   B0 13     BCS $B02E
         >.  B01B   A5 6E     LDA $6E
         >.  B01D   09 7F     ORA #$7F
         >.  B01F   25 6A     AND $6A
         >.  B021   85 6A     STA $6A
```

If the displayed addresses are in RAM, then you can change the bytes following the address. Type in your change and press <RETURN>, to make the change. The instrtuction is re-disassembled. On the next line, the following address is automatically displayed and the cursor is placed over the first byte of the instruction, so that the next instruction can be changed. This mode can be exited by erasing the character after the address before pressing <RETURN>.

4. Fill memory range >F XXXX YYYY ZZ

The area from addresses XXXX through YYYY are filled with the byte ZZ.

Example: >F 8000 8FFF 00

5. Execute program >G XXXX

The Go command executes a jump to address XXXX and executes the machine language program found there. If XXXX is not entered, the value of the program counter (PC) is used as the starting value.

If the machine language program encounters the command BRK ($00), control returns to the monitor which displays *B (break) and displays the register contents. The program counter points to the address after the BRK command. When testing programs, we recommend that you terminate them with BRK ($00).

6. Searching memory areas

There are two options when searching: search for a byte combination or search for ASCII text.

6.a Search for byte combination >H XXXX YYYY BB BB BB

The memory range from addresses XXXX through YYYY is searched for the byte combination BB. The combination can be up to 29 bytes long.

Example: >H E000 EFFF 20 D2 FF

The memory area from addresses XXXX through YYYY is searched for the combination $20 $DF $FF (subroutine call). Addresses at which this combination is located are displayed.

6.b Search for text >H XXXX YYYY "TEXT"

The memory area from address XXXX to YYYY will be searched for the ASCII text "TEXT". The text can be up to 29 characters long. Addresses at which this text is located will be displayed.

Example: H> A000 AFFF "READY"
A378

7. Load a machine language program >L "name",XX,YYYY

The program "name" is loaded beginning at address YYYY from device XX. Normally YYYY is omitted; the program then loads at the address from which it was saved. If the device address is also omitted, device 8 is assumed.

Example: >L "PROG",8
SEARCHING FOR PROG
LOADING
>

If you want to load from cassette, enter 01 for XX.

8. Display memory contents >M XXXX YYYY

The contents of memory starting at XXXX and ending at YYYY is displayed. Both XXXX and YYYY are four digit hexadecimal numbers. If the ending address YYYY is omitted, only one line is displayed. The ASCII representation of the memory contents is displayed in reverse following the hexadecimal representation. Un-printable control characters are displayed as a period.

Example: >M A0A0 A0AF
>: A0A0 C4 46 4F D2 4E 45 58 D4 DFORNEXT
>: A0A8 44 41 54 C1 49 4E 50 55 DATAINPU

Memory contents can be changed in the same way as register contents, by overwriting the byte value and pressing <RETURN>.

9. Program execution with breakpoints >Q XXXX

The single-step mode often takes too long when working with machine language programs. Therefore MONITOR 64 offers you the option of controlling machine language programs by setting breakpoints.

You can specify that a machine language program is to be interrupted when it reaches a certain place. Should the program never reach the breakpoint, it can be stopped by pressing the <RUN/STOP> key. The breakpoints are set with the U command, described shortly. The syntax of the Q command is the same as for the G and W commands.

10. Display the register contents >R

The contents of the processor registers are displayed.

The labels identifying the registers are:

PC	program counter
IRQ	interrupt vector
SR	status register
AC	accumulator
XR	X register
YR	Y register
SP	stack pointer

In addition, the flags of the status register are displayed individually:

N	negative flag
V	overflow flag
-	not used
B	break flag
D	decimal flag
I	interrupt flag
Z	zero flag
C	carry flag

Example:
```
>R
   PC  IRQ   SR AC XR YR SP NV-BDIZC
>; 0003 EA31 32 34 02 A2 F8 00110010
```

If you want to change the register contents, you simply move the cursor to the appropriate place, overwrite the old contents with the new value and press <RETURN>. The new register contents are placed into the register. If the contents of the status register are changed, the flags are also changed and displayed.

11. Save a machine language program >S "name",XX,YYYY,ZZZZ

XX is again the device address, YYYY is the starting address, and ZZZZ is the ending address plus one of the program to be saved.

Example: >S "PROG",01,C900,C9DE
SAVING PROG

The program "PROG" is saved onto cassette from address $C900 to $C9DD.

12. Transfer memory area >T XXXX YYYY ZZZZ

The memory area from addresses XXXX through YYYY are moved to the memory area beginning at ZZZZ.

Example: >T 6000 6FFF 3000

The memory range from $6000 through $6FFF is transferred to $3000 to $3FFF. The contents of the original range remains unchanged.

13. Set a breakpoint >U XXXX YYYY

If you want to use the Q command, you must first set a breakpoint. The U command performs this function. XXXX is the address at which the program is to stopped. If you start your program with the Q command, it will stop executing at the address given by XXXX. You are then placed in the single-step mode (W). With <RUN/STOP> you can halt or single-step a program. The U command offers the additional option of stopping the program after it reaches the given breakpoint a certain number of times. The YYYY parameter specifies the number of times the breakpoint is ignored before execution is halted.

Example: >U 1000 0050

Here the program is interrupted when it passes address $1000 for the 80th time (hexadecimal 50). Values up to $FFFF = 65535 are allowed.

14. Single-step mode >W XXXX

One special feature of MONITOR 64 is the single-step (walk) mode. With this you can execute machine language programs instruction by instruction. The command has the same syntax as the G command, either starting at address XXXX or at the address contained in the of program counter if only a W is given. When you enter W, the command at that address is executed and the contents of the registers and flags are displayed in the same format as with the R command. Displayed on the next line is the following

instruction in disassembled form. If you press a key, the next command is executed and the resulting register contents are again displayed. You can exit the single-step mode with the <RUN/STOP> key.

```
Example:  >W BC16
          >:  BC18 EA31 22 69 34 00 F6 00100010
          >.  BC18   86 70       STX $70
```

The single-step mode works with all "normal" programs. It should not be used with programs that use the I/O kernal functions.

15. Return to BASIC >X

The >X command returns you to Commodore BASIC. After exiting the Monitor with the X command you can enter SYS 2 or a SYS to any location containing a zero, as long as the <RUN/STOP><RESTORE> key has not been pressed in order to return to MONITOR (otherwise use SYS 12*4096).

D. ERROR MESSAGES

If you have made an error in your input, MONITOR 64 will echo the input along with a question mark. You can then correct the input.

In addition to these syntactical errors, the error routines of the kernal are activated through MONITOR 64. If an error occurs when saving or loading, for example, an error message of the following form appears:

I/O ERROR #X

in which X can be a number from 1 to 9 and has the following significance:

1 too many files
2 file open
3 file not open
4 file not found
5 device not present
6 not input file
7 not output file
8 missing filename
9 illegal device number