This is the manual for Commodore's "Assembler" program. Standard Project64 conditions and disclaimers apply. I fixed a lot of typos and errors, especially in the source code listings, so this should be a pretty good copy (sans the typos I put into the text). :)

White Flame (aka David Holz)
http://fly.to/theflame

THE COMMODORE 64 MACRO ASSEMBLER DEVELOPMENT SYSTEM

Professional Computer Division
1200 Wilson Drive
West Chester, PA 19380

COPYRIGHT

DISCLAIMER

[standard "your liability, not ours" legal tripe - wf]

PREFACE

The Commodore 64 MACRO ASSEMBLER DEVELOPMENT SYSTEM software package allows you to program in the native 6500 series Assembly language code, directly on the Commodore 64 computer. It provides you with a very powerful macro Assembler, editor, loaders and two machine language monitors along with other support routines. These development tools operate like and provide the same level of direct machine interface as the Assemblers on much larger computers.

This package contains everything that you will need to create, Assemble, load and execute 6500 series Assembly language code. You will notice

that like the software contained on this diskette, this user's manual is directed towards the experienced computer user that already has some familiarity with the 6500 series Assembly language and the operations of the Commodore 64 computer.

This product is not intended to providet he knowledge of 'how to' in assembly language, but provides the software tools for the experienced assembly language programmer.

It is recommended that the user obtain one or more of the reference manuals listed below for a more detailed description of 6502 assembly language and the Commodore 64. (The publisher is listed in parenthesis.)

- 6502 Assembly Language Subroutines, Leventhal and Saville (Osborne/McGraw-Hill)
- 6502 Software Design, Scanlon (Howard W. Sams & Co.)
- 6502 Assembly Language Programming, Leventhal (Osborne/McGraw-Hill)
- Commodore 64 Programmer's Reference Guide (Commodore/Howard W. Sams & Co.)
- Programming in 6502, Rodnay Zaks (Sybex)

This manual has been divided into five parts for easier reference. Part One, "Introduction" provides a brief descritpion of how an assembler works along with some basic terminology used throughout this manual. It is recommended that the novice user read this section first to obtain a feel for the level of knowledge needed to program in assembly language and use this manual.

Part Two, "64 Macro Assembler Capabilities and Conventions", is composed of Sections 1-4 and describes those capabilities and conventions used by this assembler.

Part Three, "Creating and Editing Assembly Source Files", is composed of Sections 5-6 and describes how to create and edit an assembly language source file. Section 5 contains the instructions for loading a support program or wedge. This program gives the user additional commands for maintaining the disk and loading and running programs. Section 6 contains the operating instructions for loading and running the Editor64 program. This program allows the user to create and edit assembly source files.

Part Four of the manual, Assembling and Testing a Program", is composed of Sections 7-9 and contains information on the programs that allow the user to assemble, test, and debug object programs. Section 7 describes the operation of the assembler program; Section 8 describes the programs that must be used to load an object program into memory; Section 9 describes the program that allows the user to monitor memory for debugging purposes.

Finally, Part Five, "Appendices", includes those charts and tables that can be used as a reference to other sections. It also provides a quick reference to the commands available when running certain programs.

USER CONVENTIONS

Throughout this manual there are certain conventions used to help make explanations less ambiguous. A list of these conventions is given below. We recommend that the user become familiar with these.

( ) Parentheses are used to denote an option. The only exceptions to this rule are in those sections where indirect indexed and indexed indirect addressing are explained. In these cases the parentheses are required.

label This is used to denote a label reference in an assembler source program. The actual label used is determined by the programmer.

opcode This is used to denote one of the 6502 instructions as specified in Appendix IV.

operand This is used to denote the operand, or argument portion of an instruction.

comments This is used to specify user comments.

filename This is used to specify a filename on disk. The actual name is determined by the user.

filename* This is used to denote a wild card filename (i.e., a filename that begins with the characters preceding the "*").

lower case variable Generally, lower case variables specify that it is up to you to supply the actual data.

UPPER CASE NAMES Generally, UPPER CASE NAMES are the actual input to be typed.

TABLE OF CONTENTS [page numbers omitted - wf]

INTRODUCTION

This manual describes the Assembly Language and assembly process for Commodore 64 programs which use one of the 6500 series microprocessors. Several assemblers are available for 6500 series program development, each is slightly different in detail of use, yet all are the same in principle. The 6500 series processors include the 6502 through the 6515 (the instruction sets are identical).

The process of translating a mnemonic or symbolic form of a computer program to actual machine code is called assembly, and a program which performs the translation is an assembler. We refer to the symbolic form of the program as source code and the actual association for those symbols are the Assembly Language. In general, one Assembly Language statement will translate into one machine instruction. This distinguishes an assembler from a compiler which may produce many machine instructions from a single statement. An assembler which executes on a computer other than the one for which code is generated, is called a cross-assembler. Use of cross-assemblers for program development for microprocessors is common because often a microcomputer system has fewer resources than are needed for an assembler. However, in the case of the Commodore 64, this is not true. With a floppy disk

and printer, the system is well suited for software development.

Normally, digital computers use the binary number system for representation of data and instructions. Computers understand only ones and zeros corresponding to an 'ON' or 'OFF' state. Users, on the other hand, find it difficult to work with the binary number system and hence, use a more convenient representation such as octal (base 8), decimal (base 10), or hexadecimal (base 16). Two representations of the 6500 series operation to 'load' information into an 'accumulator' are:

```
10101001  (binary)
A9        (hexadecimal)
```

An instruction to more the value of 21 (decimal) to the accumulator is:

```
A9 15     (hexadecimal)
```

Users still find numeric representations of instructions tedious to work with, and hence, have developed symbolic representations. For example, the preceding instruction might be written as:

```
LDA #21
```

In this example, LDA is the symbol for A9, Load the Accumulator. An assembler can translate the symbolic form LDA to the numeric form A9.

Each machine instruction to be executed has a symbolic name referred to as an operation code (opcode). The opcode for "store accumulator" is STA. The opcode for "transfer accumulator to index x" is TAX. The 56 opcodes for the 6500 series processors are detailed in Appendix IV. A machine instruction in Assembly Language consists of an opcode and perhaps operands, which specify the data on which the operation is to be performed.

A label is a 'name' for a line of code. Instructions may be labelled for reference by other instructions, as shown in:

```
L2  LDA #12
```

The label is L2, the opcode is LDA, and the operand is #12. At least one blank must separate the three parts (fields) of the instruction. Additional blanks may be inserted by the programmer for ease of reading. Instructions for the 6500 series processors have at most one operand and may have none. In these cases, the operation to be performed is totally specified by the opcode as in CLC (Clear the Carry Bit).

Programming in Assembly Language requires learning the instruction set (opcodes), addressing conventions for referencing data, the data structures within the processor, as well as the structure of Assembly Language programs. The user will be aided in this by reading and studying the 6500 series hardware and programming manuals supplied with this development package. [anybody have info on these? - wf]

1.0 INSTRUCTION FORMAT CONVENTIONS

Assembler instructions for the Commodore 64 assembler are of two basic types according to function:

- Machine instructions, and
- Assembler directives

Machine instructions correspond to the 56 operations implemented on the 6500 series processors. The instruction format is:

(label) opcode (operand) (comments)

Fields are bracketed to indicate that they are optional. Labels and comments are always optional and many opcodes such as RTS (Return from Subroutine) do not require operands. A line may also contain only a label or only a comment.

A typical instruction showing all four fields is:

```
LOOP  LDA BETA,X   ;FETCH BETA INDEXED BY X
```

A field is defined as a string of characters separated by a space.

A label is an alphanumeric string of from one to six characters, the first of which must be alpha. A label may not be any of the 56 opcodes, nor any of the special single characters, i.e. A, S, P, X or Y. These special characters are used by the assembler to reference the:

- Accumulator (A)
- Stack pointer (S)
- Processor status (P)
- Index registers (X and Y)

A label may begin in any column provided it is the first field of an instruction. Labels are used on instructions as branch targets and on data elements for reference in operands.

The operand portion of an instruction specifies either an address or a value. An address may be computed by expression evaluation and the assembler allows considerable flexibility in expression formation. An Assembly Language expression consists of a string of names and constants separated by operators, +, -, *, and / (add, subtract, multiply, and divide). Expressions are evalued by the assembler to computer operand addresses. Expressions are evaluated left to right with no operator precedence and no parenthetical grouping. Note that expressions are evaluated at assembly time and not execution time.

Any string of characters following the operand field is considered a comment and is listed, but not further processed. If the first nonblank character of any record is a semi-colon (;), the record is processed as a comment. On instructions which require no operand, comments may follow the opcode. At least one space must separate the fields of an instruction.

Appendix V presents a sample output listing from the assembler. Various examples of instruction format are included.

1.1 Symbolic

Perhaps the most common operand addressing mode is the symbolic form as in:

```
 LDA  BETA  ;PUT BETA VALUE IN ACCUMULATOR
```

In this example, BETA is a label referencing a byte in memory that contains the value to be loaded into the accumulator. BETA is a label for an address at which the value is located. Similarly, in the instruction:

```
 LDA  ALPHA + BETA
```

the address ALPHA + BETA is computed by the assembler, and the value at the computer address is loaded into the accumulator.

Memory associated with the 6500 series processors is segmented into pages of 256 bytes each. The first page, page zero, is treated differently by the assembler and processor for optimization of memory storage space. Many of the instructions have alternate operation codes if the operand address is in page zero memory. In those cases, the address is only one byte rather than the normal two. For example:

```
 LDA  BETA
```

If BETA is located at byte 4B in page zero memory, then the code generated is A5 B4. This is called page zero addressing. If BETA is at 013C, which is located in memory page one, the code generated is AD 3C 01. This is an example of 'absolute' addressing. Thus, to optimize storage and execution time, a programmer should design with data areas in page zero memory whenever possible. (Please avoid assembling code in page zero, as problems may be encountered.) Remember, the assembler makes decisions on which form to use, based on operand address computation.

1.2 Constants

Constant values in Assembly Language can take several forms. If a constant is other than decimal, a prefix character is used to specify type:

$ (Dollar sign) specifies hexadecimal
@ (Commercial at) specifies octal
% (Percent) specifies binary
' (Apostrophe) specifies an ASCII literal character in immediate instructions.

The absence of a prefix symbol indicates decimal a value. In the statement:

```
LDA  BETA + 5
```

the decimal number 5 is added to BETA to compute the address. Similarly,

```
LDA  BETA + $5F
```

denotes that the hexadecimal value of 5F is to be added to BETA for the address computation.

The immediate mode of addressing is signified by a # (pound sign) followed by a constant. For example:

```
LDA #2
```

specifies that the decimal value 2 is to be put into the accumulator. Similarly,

```
LDA #'G
```

will load the ASCII value of character G into the accumulator. Since the accumulator is one byte, the value loaded must be in the range of 0 to 255 (decimal).

Note that constant values can be used in address expressions and as values in immediate mode addressing. They can also be used to initialize locations as explained in a later section as assembler directives.

1.3 Relative

There are eight conditional branch instructions available to the user. In this example:

```
BEQ  START   ;IF EQUAL BRANCH TO START
```

If the values compared are equal, a transfer to the instruction labeled START is made. The branch address is a one byte positive or negative offset which is added to the program counter during execution. At the time the addition is made, the program counter is pointing to the next instruction beyond the branch instruction. The offset is based on the location of the next instruction. A branch address must be within 127 bytes forward or 128 bytes backward from the conditional branch instruction. An error will be flagged at assembly time if a branch target falls outside the bounds for relative addressing. Relative addressing is not used for any instructions other than branch.

1.4 Implied

Twenty-five instructions such as TAX (Transfer Accumulator to Index X) require no operand, and hence, are single byte instructions. Thus, the operand addresses are implied by the operation code.

Four instructions, ASL, LSR, ROL, and ROR, are special in that the accumulator, A, can be used as an operand. In this special case, these four instructions are treated as implied mode addressing and only an operation code is generated.

1.5 Indexed Indirect

In this mode, the operand address is computed by first adding the X register (the index) to the argument in the operand (in the example below, BETA). The resulting value is the indirect page zero address which contains the actual operand address. In this example:

```
LDA (BETA,X)
```

the parentheses around the operand indicates indirect mode. In the above example, the value in index register X is added to BETS. That sum must reference a location in page zero memory. During execution, the high order byte of the address is ignored; thus, forcing a page zero address. The two bytes starting at that location in page zero memory are taken as the address of the operand in low byte, high byte format. For purposes of illustration, assume the following:

```
BETA contains $12
X contains $04
Locations $0017 and $0016 contain $01 and $25
Location $0125 contains $37
```

Then BETA + X is $16, the address at location $16 is $0125, the value at $0125 is $37, and hence the instruction LDA (BETA,X) loads the value $37 into the accumulator. (This addressing mode is often used for accessing a table ofaddress vectors in page zero.) This form of addressing is shown in the following illustration.

```
  LDA (BETA,X)

Address    Value

      +---------+
BETA  |   $12  | + $04 = $0016
      +---------+

      +---------+
$0016 |   $25  | Treated as Low Byte
      +---------+
$0017 |   $01  | Treated as High Byte, result is $0125
      +---------+

      +---------+
$0125 |   $37  | This value is loaded into the Accumulator
```

```
      +---------+
```

1.6 Indirect Indexed

Another mode of indirect addressing uses index register Y and is illustrated by:

```
 LDA  (GAMMA),Y
```

In this case, GAMMA references a page zero location at which an address is to be found. The value in index Y is added to that address to compute the actual address of the operand. Suppose for example that:

```
 GAMMA contains $38
 Y contains $07
 Locations $0039 and $0038 contain $00 and $54
 Location $005B contains $26 which
```

The address at $38 is $0054; seven is then added to this, giving an effective address $005B. The value at $005B is $26 which is loaded into the accumulator.

In indexed indirect, the index X is added to the operand prior to the indirection. In indirect indexed, the indirection is done and then the index Y is added to compute the effective address. Indirect mode is always indexed except for a JMP instruction which allows an absolute indirect address, as exemplified by JMP (DELTA) which causes a branch to the address contained in locations DELTA and DELTA + 1. The indirect indexed mode of addressing is shown in the following illustration.

```
  LDA (GAMMA),Y

 Address    Value

      +---------+
GAMMA   |  $38  |
      +---------+

      +---------+
$0038  |  $54  | Treated as low byte
      +---------+
$0039  |  $00  | Treated as high byte, result is $0054
      +---------+ Add $07 from Y, result is $005B

      +---------+
$005B  |  $26  | This value is loaded into the Accumulator
      +---------+
```

2.0 ASSEMBLER DIRECTIVES

There are eleven assembler directives used to reserve storage and direct information to the assembler. Nine have symbolic names with a period as

the first character. The tenth, a symbolic equate, uses an equals sign (=) to establish a value for a symbol. The eleventh, asterisk (*), means the value of the current location counter. This corresponds to the ORG directive in some assemblers. It is sometimes read as "here" or "this location". Some equate examples are "RED = 5", "BLUE = $FF", and "* = $0200". A list of the directives is given below (their use is explained in this section):

```
.BYTE  .WORD  .DBYTE .PAGE  .SKIP
.OPT   .END   .FILE  .LIB
=
*
```

Labels and symbols other than directives may not begin with a period.

Examples of assembler directives can be seen in the sample Assembler program in Appendix V.

If desired, all directives which are preceded by the period may be abbreviated to the period and three characters, e.g., '.BYT'.

.BYTE is used to reserve one byte of memory and load it with a value. The directive may contin multiple operands which will store values in consecutive bytes. ASCII strings may be generated by enclosing the string with quotes. (All quotes are "single" quotes, i.e., SHIFT 7.) It should be noted, however, that there is a limitation of 40 ASCII characters that can be stored in each .BYTE directive.

```
HERE   .BYTE 2
THERE  .BYTE 1, $F, @3, %101, 7
ASCII  .BYTE 'ABCDEFH'
```

Note that numbers may be represented in the most convenient form. In general, any valid 6500 series expression which can be resolved to eight bits, may be used in this directive. If it is desired to include a quote in an ASCII string, insert two quotes in the string. For example:

```
.BYTE  'JIM"S CYCLE'
```

could be used to store:

```
JIM'S CYCLE
```

It should be noted that the use of arithmetic operations in the .BYTE directive is not supported in this version of the package.

.WORD is used to reserve and load two bytes of data at a time. Any valid expression, except for ASCII strings, may be used in the operand field. For example:

```
HERE   .WORD 2
THERE  .WORD 1, $FF03, @3
WHERE  .WORD HERE, THERE
```

The most common use for .WORD is to generate addresses as shown in the previous example labelled "WHERE", which stores the 16 bit address of "HERE" and "THERE". Addresses in the 6500 series are fetched from memory in the order low-byte, then high-byte. Therefore, .WORD generates the value in this order.

The hexadecimal portion of the example ($FF03) would be stored $03, $FF. If this order is not desired, use .DBYTE rather than .WORD.

.DBYTE is exactly like .WORD, except the bytes are stored in high-byte, low-byte order. For example:

```
 .DBYTE  $FF03
```

will generate $FF, $03. Thus, fields generated by .DBYTE may not be used as indirect addresses.

An advanced technique is to set up vector tables under the assembler directive .DBYTE and to push the starting vector address onto the stack, then execute an RTS instruction to access your routine. Remember that the addresses for the operands should be the actual address location minus one. When constructing a JUMP table in the usual way using an indirect jump instruction (opcode JMP ($6C)), do not subtract one from the address of the operand.

Equal (=) is the EQUATE directive and is used to reserve memory locations, reset the program counter (*), or assign a value to a symbol.

```
HERE   * = * + 1  ;RESERVE ONE BYTE
WHERE  * = * + 2  ;RESERVE TWO BYTES
* = $200         ;SET PROGRAM COUNTER
NB = 8           ;ASSIGN VALUE
MB = NB + %101    ;ASSIGN VALUE
```

The '=' directive is very powerful and can be used for a wide variety of purposes.

Asterisk (*) directive is used to change the program counter. To create an object code program that starts assembly at any address greater than zero, the '*' directive must be used. For example, '* = $200' starts assembling at address $200.

Expressions must not contain forward references or they will be flagged as an error. For example:

```
 * = C + D - E + F
```

would be legal if C, D, E and F are all defined, but would be illegal if any of the variables were defined later on in the program. Note also that expressions are evaluated in strict left to right order.

.PAGE is used to cause an immediate jump to the top of page on the output listing and may also be used to generate or reset the title printed at the top of the output listing.

```
.PAGE   'THIS IS A TITLE'
.PAGE
.PAGE   'NEW TITLE'
```

If a title is defined, it will be printed at the top of each page until it is redefined or cleared. A title may be cleared with:

```
.PAGE   ' '
```

.SKIP is used to generate blank lines in a listing. The directive will not appear, but its position may be found in a listing. The directive is treated as a valid input "list" and the list number printed on the left side of the listing will jump by two when the next line is printed.

```
.SKIP  2          ;SKIP TWO BLANK LINES
.SKIP  3 * 2 - 1  ;SKIP FIVE LINES
.SKIP             ;SKIP ONE LINE
```

.OPT is the most powerful directive and is used to control the generation of output fields, listings and expansion of ASCII strings in .BYTE directives. The options available are: ERRORS, NOERRORS, LIST, NOLIST, GENERATE, NOGENERATE, SYM, and NOSYM.

```
.OPT   ERRORS, LIST, GENERATE
.OPT   NOE, NOL, NOG
```

Also valid is:

```
.OPT   LIST, ERR
```

Default Settings are:

```
.OPT   LIST, ERR, NOGEN
```

Here are descriptions for each of the options:

ERRORS, NOERRORS:
Used to control creation of a separate error file. The error file contains the source line in error and the error message. This facility is normally of greatest use to time-sharing users who have limited print capacity. The error file may be turned on and examined until all errors have been corrected. This listing file may then be examined. Another possibility is to run with:

```
.OPT   ERROR, NOLIST
```

until all errors have been corrected, and then make one more run with:

```
.OPT   NOERRORS, LIST
```

LIST, NOLIST:
Used to control the generation of the listing file which contains source input, errors/warnings, code generation, symbol table and

instruction count if enabled.

GENERATE, NOGENERATE:
Used to control printing of ASCII strings in the .BYTE directive. The first two characters will always be printed, and subsequent characters will be printed (normally two bytes per line), if GENERATE is used.

.END should be the last directive in a file and is used to signal the physical end of the file. Its use is optional, but highly recommended for program documentation.

.LIB allows the user to insert source code from another file into the assembly. When the assembler encounters this directive, it temporarily ceases reading source code from the current file and starts reading from the file named in the .LIB. Processing of the original source file resumes when end-of-file (EOF) or .END is encountered in the library file. The control file containing the .LIB can contain other assembler directives to turn the listing function on and off, etc.

.FIL can be used to link another file to a current one during assembly. A library file called by a .LIB may not contain another .LIB, but it may contain a .FIL. A '.FIL' terminates assembly of the file containing it and transfers source reading to the file named on the OPERAND. There are no restrictions on the number of files which may be linked by .FIL directives. Caution should be exercised when using this directive to ensure that no circular linkages are created. An assembler pass can only be terminated by (EOF) or the .END directive.

3.0 MACRO CAPABILITIES

Macros take the general form shown in the following code:

```
(label)  .MAC  macro name
         TEXT OF MACRO
(label)  .MND
```

The directive .MAC defines a macro with the given macro name and creates up to nine parameters for that macro. The user does not explicitly declare parameters. Macros may contain arbitrary text, except that they cannot contain the directives .MAC and .MND. The directive .MND identifies the end of a macro definition. The labels on .MAC and .MND are optional as denoted by the parentesis. They respectively label the first generated statement and the statement immediately succeeding the last generated statement.

To call a macro, the user simply gives the macro name and lists the parameters as indicated in the call line below:

```
macroname  param1, param2, ...
```

The macro name must be delimited by a space before the first parameter as indicated above.

Within the text of the macr definition, a parameter is designated by the temporary symbol "?" followed by a digit 1 through 9. Thus, '?3' designates parameter 3. During assembly, if a macro call is encountered, the text included in that macro is inserted into the assembly at that point and the parameter names at the calling point are substituted for the temporary names. If the user fails to supply a parameter name when the macro is called, the assembler will generate a name for that parameter (if one is needed) for the duration of that call.

To give a brief example, suppose we wish to increment a double precision (16-bit) quantity. Then, the macro definition to do this is:

```
   .MAC   DPINC     ;DOUBLE PRECISION INCREMENT
    INC   ?1
    BNE   ?2
    INC   ?1+1
?2  .MND
```

When the macro is called to increment the variable COUNT, the following call line is used.

```
   DPINC   COUNT
```

This generates the following code:

```
   INC   COUNT
   BNE   L001
   INC   COUNT+1
L001
```

In this example, the internal label name "L001" is generated automatically during macro expansion. Subsequent calls produce distinct labels following the progression L002, L003, etc. If the programmer supplies a second parameter in the calling line, instead of leaving that parameter blank, the internal label name will be set to the second parameter instead of L001.

Macros can call other macros, but the depth of the nesting cannot exceed eight levels.

Empty Parameters

Empty parameters in call lines are denoted by commas:

```
  FUNCTN  AA,,,DD     ;PARAMETERS ?2 AND ?3 ARE EMPTY AS
                      ;ARE ?5 THROUGH ?9
  FUNCTN  ,,CC,,EE    ;?1, ?2, AND ?4 ARE EMPTY AS ARE
                      ;?6 THROUGH ?9
```

The calls on FUNCTN given here will result in different internal local parameters being given generated names, or names supplied from the programmer.

Concatenated Names

A macro parameter is supplied to a macro without leading or trailing blanks, so that a parameter can be used to create new variable names and allocate space for the variables.

```
    .MAC   DECL     ;DECLARED STORAGE
XX?1  .WOR   0
      * = * + ?2
    .MND

     DECL  AA,5     ;THIS IS A MACRO CALL
XXAA  .WOR   0
     * = * + 5      ;THIS IS HOW IT EXPANDS

     DECL  A2,10    ;THIS IS A SECOND CALL
XXA2  .WOR   0
     * = * + 10     ;THIS IS HOW THE SECOND CALL EXPANDS
```

Notice that there are no blanks in the labels XXAA and XXA2.

Expressions As Parameters

Parameters of macros can be arbitrary expressions that do not include embedded commas, semicolons, or blanks. Whent he expressions are inserted into the macro definition, the expression must make sense to the assembler.

```
 .MAC   LSS         ;IF ACCUM LESS THAN ?1 GOTO ?2
  CMP   ?1
  BCS   ?2
 .MND

  LSS   XX+5,EXIT    ;COMPARE WITH LOCATION XX+5
  LSS   #$F3,EXIT    ;COMPARE WITH LITERAL
  LSS   (XX,1),EXIT  ;ILLEGAL--AN EMBEDDED COMMA
```

Assembler Output Format

Note: The macro assembler uses different rules than the previous Commodore assembler in deciding how to format a print line. The new rule is the following:

> An identifier that begins in column 1 is printed as a label, otherwise it is assumed to be an opcode.

This rule is identical to the rule used by the editor for the FORMAT command. Hence, the FORMAT command now permits the user to view the final printed format of edited files.

4.0 OUTPUT FILES GENERATED BY THE ASSEMBLER

There are three output files genereated by the assembler. Each file is

optional and can be created through the use of the .OPT assembler directive. The listing file contains the program list with errors and the symbol table. The error file contains all error lines and errors (as included in the listing file). The interface file contains the object code for the loader.

The cross reference file may be optionally be generated by the assembler. This file is used by the cross reference program to print a report showing all variables, their declared addresses, and all line numbers in which each variable is used.

Listing File

The listing file will be produced unless the NOLIST option is used on the .OPT assembler directive. This file is make up of two sections: Program and Error List, and Symbol Table.

o Program and Error List

This listings will always be produced unless the NOLIST option is selected. It contains the source statement of the program along with the assembled code. Errors and warning appear after erroneous statements. (An explanation of error codes is presented in Appendix VI.) A count of the errors and warnings found during the assembly is presented at the end of the program.

o Symbol Table

The symbol table will always be produced unless the NOSYM option is used. It contains a list of all symbols used in the program, and their addresses.

Interface File

This file does not contain true object code, but data which can be loaded and converted to machine code by the loader. The format for the first and all succeeding records, except for the last record, is as follows:

; n1n0 a3a2a1a0 (d1d0)1 (d1d0)2 ... (d1d0)23 x3x2x1x0

Where the following statements apply:

1. All characters (n,a,d,x) are the ASCII characters zero through F, each representing a hexadecimal digit.

2. The semicolon is a record mark indicating the start of a record.

3. n1n0 The number of bytes of data in this record (in hexadecimal). Each pair of hexadecimal characters (d1d0) represents a single byte.

4. a3a2a1a0 The hexadecimal starting address for the record. The a3 represents address bits 15 through 12, etc. The 8-bits

represented by (d1d0)1 is stored in address a3a2a1a0;
(d1d0)2 is stored in (a3a2a1a0)+1, etc.

5. (d1d0) Two hexadecimal digits representing an 8-bit byte of data. (d1=high-order 4 binary bits and d0=low-order 4 bits). A maximum of 18 (hex) or 24 (decimal) bytes of data per record is permitted.

6. x3x2x1x0 Record check sum. This is the hexadecimal sum of all characters in the record, including the n1n0 and a3a2a1a0, but excluding the record mark and the check sum of characters. To generate the check sum, each byte of data (represented by two ASCII characters) is treated as 8 binary bits. The binary sum of these 8-bit bytes is truncated to 16 binary bits (4 hexadecimal digits) and is then represented in the record as four ASCII characters (x3x2x1x0).

The format for the last record in a file is as follows:

; 00 c3c2c1c0 x3x2x1x0

1. ; 00 Zero bytes of data are in this record. The zeros identify this as the final record in a file.

2. c3c2c1c0 This represents the total number of records (in hexadecimal) in this file, NOT including the last record.

3. x3x2x1x0 Check sum for this record.

5.0 ADDITION BASIC DISK COMMANDS (DOS SUPPORT WEDGE)

On the release disk is a program which will aid you in performing disk housekeeping functions (copying, scratching, renaming, reading the directory, intializing the disk drive, checking the disk status, and loading (and running) programs from disk). The commands that this program provides are short and simple and are very useful.

5.1 Loading the DOS WEDGE64 Program

When this program is loaded and executed, it "wedges" itself into the operating system and BASIC interpreter. Thus, the wedge checks all keyboard entries for its command characters before passing the entry on to the BASIC interpreter. (This is done by linking into the CHRGET routine in page zero).

To load the wedge program, enter:

LOAD "DOS WEDGE64",8

and press RETURN. This will load a program that "boots" the actual wedge program into memory. Once loaded, type RUN and press RETURN

before removing the diskette. When the wedge program is loaded, a copyright notice will be displayed.

5.2 Using the DOS WEDGE64 Program

The wedge program supports all of the same commands that are included in BASIC (copy, scratch, rename, new a disk), a command to read the directory (without overwriting memory), and commands to load and run programs. The wedge program also provides you with the capability of creating and maintaining volumes of files (volume creation allows you to group certain programs together) and the capability to perform operations using a wild card filename (a file whose name begins with certain characters).

Each command begins with a single character as specified in Section 5.3. The character used depends on the command. The @ (commercial at sign) and > (greater than sign) are used interchangeably to begin any of the disk housekeeping commands or to read the directory. They are also used to reset or initialize the drive, and to terminate the DOS Wedge. The ^ (up arrow) is used to begin the command to load (at BASIC's Start of Text address) and automatically run a program. The / (slash) is used to begin the command to load a program at BASIC's Start of Text address. The % (percent sign) is used to begin the command to load a program at its load address. Finally, the <- (back arrow) is used to begin the command for saving a file to disk.

5.3 DOS WEDGE64 Program Commands

A description of each command is given in the following pages. Appendix IX provides a brief summary of the DOS WEDGE64 commands.

@

Typing this character alone will provide the user with the current disk status. This performs the same function as the following BASIC code:

```
10 OPEN 15,8,15
20 INPUT#15,A,B$,C,D
30 PRINT A;B$;C;D
40 CLOSE 15
```

@$(drive):(filename)(*)([volume)

This command will read the directory from the disk drive specified and print it to the screen. If filename is specified, only that file, if present, will be displayed. If * is specified, all files whose names begin with the letters specified by filename will be printed. If volume is specified (where volume is the character id of that volume), then only those files contained on that volume will be printed.

@N(drive):diskname,id

This command will format a disk using the name and id specified.

@R(drive):newfile([volume])=oldfile([volume])

This command will rename the file specified by oldfile to the name specified by newfile.

@C(drive):newfile([volume])=oldfile([volume])

This command will copy the file specified by oldfile to the name specified by newfile. If [volume] is specified, the newfile will be created on that volume.

@S(drive):filename(*)([volume])

This command scratches the file specified by filename. If * is specified, all files beginning with the letters specified by filename will be scratched. If [volume] is specified, only those files that are contained on that volume will be scratched.

@UI(drive)

This command will reset the DOS.

@I(drive)

This command will intialize the disk drive.

@Q

This command will terminate the wedge program.

/filename

This command will load the file specified by filename. For example:

/ASM.C64

will cause the program named "ASM.C64" to be loaded into memory. This command does the same thing as the BASIC command:

LOAD"ASM.C64",8

Please note that this command can only be used to load BASIC programs, or machine code programs that are booted from BASIC. This is because the computer will ignore the file's own load address and will instead load at the current "Start of BASIC Text" area.

%filename

This command will load the file specified by filename at its own load address. It does the same thing as the BASIC command:

LOAD "filname",8,1

where filename is the name of the program to load.

^filename

This command allows the user to load and run the program specified by filename and does the same thing as entering:

LOAD "filename",8

followed by the BASIC command RUN.

Again, please note that this command can only be used to load and run BASIC programs, or machine code programs that are booted from BASIC.

<-filename [that's the single character left-arrow - wf]

This command saves the program specified by filename to disk.

6.0 CREATING AND EDITING A SOURCE FILE

The editor is used to enter and modify source files for the assembler. The editor retains all of the features of the BASIC screen editor and allows AUTOmatic line numbering, FIND, CHANGE, DELETE within a range, and reNUMBER. Other commands include GET, PUT, BREAK, KILL, and FORMAT. All of the commands are detailed in the summary at the end of this section.

The editor commands operate in a similar fashion to the commands already existing in the computer's BASIC. For practice, we suggest that you try to create short example files using the editor commands.

The data files on which the assembler operates are made up of CBM ASCII characters with each line terminated by a carriage return. The only restriction on data lines is in naming. Due to the method in which the assembler parses, spaces are not allowed in filenames. The files are sequential and must be terminated by a zero byte $00. When listing a directory, these files will show as file type SEQ.

Each file's format is sequential, with a terminating zero byte ($00).

6.1 Loading the Editor64 Program

The editor must be loaded with the BASIC LOAD command:

LOAD "EDITOR64",8,1
or %0:EDITOR64 (if the wedge is enabled)

To intiate the editor, type 'SYS49152'. After typing the SYS command, the editor has been loaded. At this point, type a NEW command to clear the text pointers. You are now ready to eit or enter assembler source files.

6.2 Using the Editor64 Program

When the Editor64 Program is in operation, any BASIC statement typed such as:

```
10 FOR I = 1 TO 10
```

will not be tokenized (converted into BASIC keyword tokens). Thus, you cannot type a BASIC line with the editor turned on. To avoid this problem, disable the editor with the 'KILL' command or reset the computer to return to BASIC.

Source files are loaded with the 'GET' command. As the file is loaded, the editor generates the line numbers automatically starting at 1000. After editing the file, ensure that the last line in the file is a .FILE or a .END assembler directive. Then, save the file on the disk with the 'PUT' command.

Important: Be sure to save your completed file using the PUT command BEFORE loading the assembler or your file will be lost.

Refer to Appendix VII for an Editor64 Command Summary.

6.3 Editor64 Program Commands

AUTO Line Numbering

The AUTO command generates new line numbers while entering a new source code file. To enable the AUTO command, type the following:

```
AUTO n1
```

where n1 is the optional increment between line numbers printed. To disable the AUTO function, type the AUTO command without an increment.

CHANGE string

The CHANGE command automatically locates and replaces one string with another (multiple occurrences). This command is entered in the following format:

```
CHANGE/str1/str2/(,n1-n2)
```

- / Delimits the str1 and str2 (use any character not in either string)
- str1 Search string
- str2 Replacement string
- n1-n2 Range parameters. The format is the same as the LIST command in BASIC. If omitted, the whole file is searched. (Optional)

CPUT Command

The CPUT command output source files with no unnecessary spaces to the disk for later assembly. The syntax for this command is the same as the PUT command.

DELETE

The DELETE command allows the user to delete several lines at a time. Simpy input the range of lines to be deleted (n1 through n2). (The format is the same as the LIST command in BASIC).

DELETE n1-n2

To delete a single line, enter the line number alone on a blank line and press RETURN.

FIND string

The FIND command is used to search for and locate specific character strings in text. Each occurrence of the scring is printed on the CRT. You can pause the printing with the space bar. Printing can then be continued with the space bar, or terminated with the RUN/STOP key. The format of the FIND command is:

FIND/str1/(,n1-n2)

/ Delimiter (use a character not in the string)
str1 Search string
n1-n2 Range parameter. Same as the LIST command in BASIC. (Optional)

FORMATted Print

The FORMAT command is used to print the text file in tabbed format like the assembler. For this function to work correctly, you must type mnemonics in column two, or one space from labels.

FORMAT (n1-n2)

n1-n2 Range parameters of the same format as LIST. (Optional)

Note: This command has the same controls as FIND. For example, press space bar to halt printing and press it again to restart printing.
Press the RUN/STOP key to terminate the listing.

GET Command

This command is used to load assembler source text files into the editor from disk. It can also be used to append to files already in memory.

GET "filename"(,n1)(,n2)(,n3)

n1 Begins inputting source at this line in the file currently in

memory (Optional)
n2 Device number, default is 8 (Optional)
n3 Secondary address, default is 8 (Optional)

Note: GET starts numbering lines at 1000 and incrementing the line numbers by 10. If n1 is greater than any line number in memory, the file being loaded is appended to the end of the current file.

KILL Command

This command causes the editor to disengage. To restart the editor, type the same command used to start the editor (SYS49152).

LIST Command

The editor LIST command works in the same manner as the LIST command in BASIC.

LIST (n1)-(n2)

where n1-n2 specifies a range of lines. Valid parameters also include 'n1-' (which will list all lines from n1 to the end) and '-n2' (which will list all lines from the beginning up to and including n2).

ReNUMBER Lines

The NUMBER command allows the user to renumber all or part of the file in memory.

NUMBER (n1),(n2),(n3)

n1 Old start line number (Optional)
n2 New start line number (Optional)
n3 Step size for resequence (Optional)

PUT Command

The PUT command outputs source files to the disk for later assembly. PUT has the ability to output all or part of the memory resident file.

PUT "filename",(n1-n2),(n3),(n4)

n1 Starting line number (Optional)
n2 Ending line number (Optional)
n3 Device number, default is 8 (Optional)
n4 Secondary address, default is 8 (Optional)

If n1-n2,n3,n4 are left out, the whole file is output to the disk.

7.0 ASSEMBLING A SOURCE FILE

Once a source file is ready to assemble, you must first save it on disk (by using the PUT command). Please be sure to do this before loading

the assembler program. Once this is completed, you will load the assembler which will reside in the same area that BASIC programs do.

7.1 Loading the Assembler64 Program

To load the assembler, type:

```
LOAD "ASSEMBLER64",8
```
or /ASSEMBLER64 (if the DOS Wedge is loaded)

After loading is complete, type RUN and press RETURN. The assembler will print a copyright notice and the first user prompt when execution begins.

7.2 Using the Assembler64 Program

When a program is being assembled, the user has the option of creating two types of files. The first type is an object file which contains the data necessary to create a machine code program (by the loader). The name of this file is specified by the user before assembly starts. The remaining files are cross reference files. The names for these files are automatically generated by the assembler and are in the format "XXLL0000" and "XXFF0000".

It should be noted, however, that the assembler program will not overwrite any of these files. If you wish to use the same object filename each time you assemble a program, you must "scratch" the old object file before you run the assembler. In the case of cross reference files, the same procedure should be followed if you want to create new cross reference files.

Although you will be given the option of creating both object and cross reference files before assembly starts, only one of these options can be chosed (because of the number of files open at one time). If you want both files, run the assembly once with the object file option, and once with the cross reference option.

When the assembler starts, the first prompt will be:

```
OBJECT FILE (CR OR D:NAME):
```

If you want the assembler to create an object file enter the filename and press RETURN. If not, press RETURN.

Next you will be prompted with:

```
HARD COPY (CR/Y OR N)?
```

If you want a hardcopy printout, enter Y and press RETURN or simply press RETURN. If not, enter N and press RETURN. This will cause the output to be listed to the screen.

Next, you will be prompted with:

CROSS REFERENCE (CR/NO OR Y)?

If you want a cross reference file created, enter Y and press RETURN.
If not, simply press RETURN.

Finally, you will be prompted with:

SOURCE FILE NAME?

Enter the name of the source file that you wish to assemble.

After entering this last prompt, the assembler program begins to execute. If, during this assembly, the symbol table overflows, the assembly process will stop.

HALTING THE ASSEMBLER

When the assembler is running, operation may be halted by pressing the RUN/STOP key. If this is done, the assembly process will be stopped and the program will wait for the user to either continue the assembly or to terminate it completely. Press the B key to terminate the assembly and return to BASIC. Pressing any other key will continue the assembly process. This feature is useful for users without printers, as the screen listing can be examined during assembly.

CROSS REFERENCE FILES

If you choose to create a cross reference, two files will be created as was mentioned above. To look at or produce a hardcopy printout of this cross reference, you must first load the cross reference listing program. To load this program, type:

LOAD "CROSSREF 64",8

Once this program is loaded, type RUN and press RETURN and the cross reference listing program will prompt with:

HARD COPY (CR/Y OR N)?

Press RETURN if you want a hardcopy printout; otherwise, enter N and press RETURN.

8.0 LOADING AN OBJECT FILE

The Commodore 64 Assembler produces protable output in an ASCII format that can not be directly executed. This output must be LOADED so the program can be executed. This is the function of a Loader.

8.1 Loading the Loader Program

There are two versions of the loader included on the development disk. Each version is positioned in a different area of RAM memory. This allows the user to load anywhere in RAM by using the correct loader. To load one of the Loader programs, type:

```
LOAD "filename",8,1
```

where filename is the program to be loaded. The following table shows the names, load points, and run commands for each loader.

Name	Load Address	Run Command
LO-LOAD.C64	$0800	RUN
HI-LOAD.C64	$C800	SYS51200

8.2 Using the Loader Programs

Both the HI-LOAD and LO-LOAD loaders are about 512 bytes long and operate in the same manner. When activated, the loaders print a copyright notice and prompt the user for a load offset. The offset is used to place object code into an address range other than the one that it was assembled into. This allows the user to assemble for an area where there is no RAM and load into a RAM area. The object can then be programmed into EPROM, etc.

The offset is a two byte hexadecimal address that is added to the program addresses. If the program address plus the offset is greater than $FFFF, the address wraps around through $0000. The following examples show how offset works.

Program Address	Offset	Address of Object Code
$0400	$0000	$0400
$3000	$0000	$3000
$0400	$2000	$2400
$9000	$9000	$2000
$E000	$4000	$2000

After the offset is entered, the loader will prompt the user for the object filename to be loaded. The loader will then initialize the drive, search for the file, and start the load. As the data is laoded, the program will print the input data to the CRT. This is for user feedback only. When the load is completed, the loader prints the message 'END OF LOAD' and returns to BASIC.

There are three errors that can occur during a load (each is self documenting):

```
BAD RECORD COUNT
NON-RAM LOAD
CHECKSUM ERROR
```

Errors are considered fatal; the load is terminated, the object file is closed, and control is returned to BASIC.

8.0 TESTING AND DEBUGGING WITH THE MONITOR PROGRAMS

The MONITOR is the machine language monitor for the Commodore 64. This programming aid contains many features that will enable you to create, modify and test machine language programs and subroutines. The MONITOR's purpose is to make it easy for you to examine and change memory while debugging your program.

9.1 Loading the MONITOR Programs

There are two machine language monitors on the Commodore 64 Assembler Development disk: MONITOR$8000 and MONITOR$C000. The only difference is the area of memory in which the program resides. MONITOR$8000 resides at memory location $8000 and MONITOR$C000 resides at memory location $C000. The two MONITOR progarms are both included in case one interferes with the intended location for the machine code program to reside.

To load and activate the appropriate monitor, enter:

```
LOAD "MONITOR$8000",8,1 (monitor at $8000)
SYS32768
```

or

```
LOAD "MONITOR$C000",8,1 (monitor at $C000)
SYS49152
```

9.2 Using the MONITOR Programs

The MONITOR programs will respond by displaying the CPU registers, typing a period, and flashing the cursor. The period is a prompt that lets you know the MONITOR program is waiting for your command. The commands are described on the following pages. Appendix VIII provides a summary of MONITOR commands.

To exit the MONITOR program, reset the machine. An alternate method is as follows:

1) Type the following line to escape the monitor:
   .A 00FB JMP ($FFFC) (and press RETURN twice).
   The first RETURN displays .A 00FE. The second RETURN displays just the period prompt (.).
2) Now type G 00FB and press RETURN. This command causes a cold start (reset) and displays the initial startup message when you first turn on the computer. This allows you to escape the monitor and keep it in memory without turning off the computer. Now you can program in BASIC, however BASIC programs are lost once the monitor is reactivated with the appropriate SYS command appearing on page 29.

9.3 MONITOR Program Commands

COMMAND: A (ASSEMBLE)

Purpose: Enter a line of assembly code.
Syntax: A(address)(opcode mnemonic)(operand)

(address): A four-digit hexadecimal number indicating the location in memory to place the opcode.

(opcode mnemonic): A standard MOS assembly language mnemonic, i.e. LDA, STX, ROR, etc. as defined in Appendix IV.

(operand): The operand, when required, can be of any of the legal addressing modes. (For zero-page modes, a two digit hex number is required whose value is less than or equal to $FF. For non-zero page addresses, a four digit hex number whose value is less than or equal to $FFFF is required.)

A RETURN is used to indicate the end of the assembly line. If there are any errors on the line, a question mark is diplayed to indicate an error, and a period is typed on the next line. The screen editor can be used to correct any errors on the original line.

After a line of code is successfully assembled, the assembler will print a prompt containing the next legal memory location for an instruction, so 'A' and the line number do not have to be typed more than once when typing assembly language programs into the Commodore 64. To exit this mode, press RETURN after the 'A' prompt.

Example: .A 1200 LDX #$00
.A 1202

COMMAND: C (COMPARE)

Purpose: Compare two areas of memory.
Syntax: C(start address)(end address)(with address)

(start address): A four digit hex number indicating the start address of the area of memory to compare against.

(end address): A four digit hex number indicating the end address of the area of memory to compare against.

(with address): A four digit hex number indicating the start addresses of the other area of memory to compare with.

The address fields should be separated by a valid delimiter, such as a space or comma. If the two areas of memory are the same, then 64MON will print a period, indicating that the second area of memory is the same as the first. The addresses, of any bytes in the two areas which are different, are printed out on the screen in descending order.

COMMAND: D (DISASSEMBLE)

Purpose: Disassemble machine code into assembly language mnemonics and operands.
Syntax: D(address 1)(address 2)

(address 1): A four-digit hexadecimal starting address of the code to be disassembled.

(address 2): An optional four-digit hexadecimal ending address of the code to be disassembled.

The address fields should be separated by a delimiter such as a space or comma. The format of the disassembly is only slightly different than the input format of an assembly. The difference is that the first character of a disassembly is a comma, rather than an 'A' (for readability).

A disassembly listing can be modified using the screen editor. Make any changes to the mnemonic or operand on the screen, then press RETURN. This will enter the line and call the assembler for further modifications.

A disassembly can be scrolled up or down on the screen via cursor control. When a line of desassembly is at the bottom of the screen, a cursor down will cause the screen to scroll up one line to display another disassembled line of code. This also works for scrolling backwards through a disassembly (i.e., going to the top of the screen and hitting cursor up).

Example: D 1000 1400
., 1000 LDA #$00
., 1002 ???
., 1003 BNE $1030

COMMAND: F (FILL)

Purpose: Fill a range of locations with a specified byte.
Syntax: F(address 1)(address 2)(byte)

(address 1): The first location to fill with the value specified by (byte).

(address 2): The last location to fill with the value specified by (byte).

(byte): A two digit hexadecimal number to be written into consecutive memory locations.

This command is useful for intializing data structures or any other RAM area.

Example: F 0400 0518 EA

Fills memory locations from $0400 to $0518 with $EA (a NOP instruction).

COMMAND: G (GO)

Purpose: Begin execution of a program at a specified address.
Syntax: G(address)

(address): An optional argument specifying the new value of the program counter and address where execution is to start. When the address is left out, execution will begin at the current PC. (The current PC can be viewed using the R command).

The GO command will restore all registers (displayable by the R command) and begin execution at the specified starting address. Caution is recommended in using the GO command. (It may sometimes be wise to set a breakpoint somewhere int he line of program execution to prevent loss of control of the operating system.)

Example: G 040C

Execution begins at location $040C.

COMMAND: H (HUNT)

Purpose: Hunt through memory within a specified range for all occurrences of a set of bytes.
Syntax: H(address 1)(address 2)(data)

(address 1): Beginning address of hunt procedure.

(address 2): Ending address of hunt prodecure.

(data): Data set to be searched (data may be hexadecimal or an ASCII string).

An ASCII string is specified by preceding the first character with a single quote, i.e., 'STRING. Data may be single or multiple arguments. Multiple two-digit hex arguments must be separated by a space.

```
Example:  H C000 FFFF 'READ     ;SEARCH FOR ASCII STRING READ
          H A000 A101 A9 FF 4C  ;SEARCH FOR DATA $A9, $FF, $4C,
                                ;IN THAT SEQUENCE
```

COMMAND: I (INTERROGATE)

Purpose: Display memory in ASCII character format within the specified address range.
Syntax: I(address 1)(address 2)

(address 1): Starting address of ASCII dump.

(address 2): Ending address of ASCII dump.

The ASCII characters are displayed in reverse video (to contrast the

charcter with the hexadecimal data displayed on the screen).

The display can be made to scroll by using the cursor up/down key. This allows continuing the search beyond the search parameters. When a line of the listing is at the bottom of the screen, a cursor down will cause the screen to scroll up one line to display another line of the listing. This also works for scrolling backwards (i.e., going to the top of the screen and typing cursor up).

Note: When a charcter is not printable, it will be displayed as a period (.).

Example: I C000 C020

Displays in REVERSE all data from $C000 to $C020.

COMMAND: L (LOAD)

Purpose: Load a file from cassette or disk.
Syntax: L"filename"(,device)

filename: Any legal Commodore 64 filename.

(device): A two-digit byte indicating the device number from which to load.

01 is cassette
08 is disk (or 09, etc.)

The LOAD command causes a file to be loaded into memory. The starting address is contained in the first two bytes of the file (in a PRG file). In other words, the LOAD command always loads a file into the same place it was saved from. This is very important in machine language work, since few programs are completely relocatable. The file will be loaded into memory until the end of file marker (EOF) is found.

```
Example:   L"SCREEN",01   ;READS A FILE FROM CASSETTE
           L"TANK",08     ;READS A FILE FROM DISK DRIVE
```

COMMAND: M (MEMORY DISPLAY)

Purpose: To display memory as a hexadecimal dump within the specified address range.
Syntax: M(address 1)(address 2)

(address 1): First address of hex dump.

(address 2): Last addrses of hex dump. (Optional. If omitted, eight bytes will be displayed.)

Memory is displayed in the following format:

```
.:A048 7F E7 00 AA AA AE 02 FF
```

Memory content may be edited using the screen editor. To edit, move the cursor to the data to be modified. Type the desired correction and press RETURN. If there is a bad RAM location or if an attempt to modify ROM has occurred, an error flag (?) will be displayed.

As with the DISASSEMBLY and INTERROGATE commands, the screen may be scrolled both up and down by using the cursor controls.

Example: M 0000
.:0000 4C 7F EF AA 00 02 F7 FF

The first eight bytes of memory are displayed.

COMMAND: N (NEW LOCATOR)

Purpose: To relocate absolute memory references by adding an offset to the operands of the target code.
Syntax: N(address 1)(address 2)(offset)(ref 1)(ref 2)W

(address 1): Starting address of code to be modified.

(address 2): Ending address of code to be modified.

(offset): Value to be added to operand of instructions.

Code moved from a high location to a low location inmemory needs a value which wraps around. For example, a piece o code moved from $A000 to $0400 will require an offset of $6400. $A000+$6400=$10400, but since there is not a seventeenth bit in the computer, the result is $0400.

(ref 1): Any three byte instruction whose operand is greater than or equal to (ref 1) an less than (ref 2) will be offset by the (offset) value, i.e., the operand of the three byte instruction will be replaced by 'operand + (offset)'.

(ref 2): Upper limit of operands to relocate (see ref 1). Any operand with a value greater than or equal to (ref 2) will not be relocated.

W: Relocate word tables, (optional). Every two bites will be offset if the W is included. Relocation then becomes data independent.

Often it is useful to move a section of code from one area in memory to another (see the "T" command) to make room for more code. Then by using the "N" command, the code can be changed to run in the new address space.

COMMAND: R (REGISTER DISPLAY)

Purpose: Show important 6502 registers. The program status register, program counter, the accumulator, the X and Y index registers and the stack pointer are displayed.
Syntax: R

Note that the stack pointer is displayed without its implied ninth bit.

Since the ninth bit of the stack pointer has been mentioned, it is appropriate to point out a bug in the 6502. When a PHP instruction is executed, the ninth bit of the stack pointer is OR'd into the status byte and is stored on the stack with bit four (the break flag!) always set. For 99.9% of all applications, this makes no difference. However, when this bug does turn up, it causes problems which are very difficult to track down.

Example: R

```
          PC   SR  AC XR YR SP
        .;  057F  01  02 03 04 FE
```

COMMAND: S (SAVE)

Purpose: Save the contents of memory onto tape or disk.
Syntax: S"filename",(device),(address 1),(address 2)

filename: Any legal filename for saving the data. The filename must be enclosed in double quotes; single quotes are illegal.

(device): Two possible devices are cassette and disk. To save on cassette, use device 01. The device number of the Commodore disk drive is usually 08.

(address 1): Starting addrses of memory to be saved.

(address 2): Ending address of memory to be saved, plus one. All data up to, but not including the byte of data at this address, will be saved.

The file created by this command is a load file, i.e., the first two bytes contain the starting address (address 1) of the data. The file may be recalled using the 'L' command.

Example: S"GAME",08,0400,0C00

saves memory from $0400 to $0C00 onto disk.

COMMAND: T (TRANSFER)

Purpose: Transfer segments of memory from one memory area to another.
Syntax: T(address 1)(address 2)(address 3)

(address 1): Starting address of data to be moved.

(address 2): Ending address of data to be moved.

(address 3): Starting address of new location (where the data will be placed).

Data can be moved from low memory to high memory or vice-versa. Additional memory segments of any length can be moved forward or backward any number of bytes, i.e., shifted. It only transfers the machine code to the new address specified in the command. The TRANSFER

command DOES NOT relocate absolute addresses for you. If absolute addressing is used, the relative offset of the original addrses IS NOT calculated for the value in the operand field. You must manually go back and change any absolute addresses.

Example: T 1400 1600 1401

shifts data from $1400 up to and including $1600, one byte higher in memory.

APPENDIX I OPERATING SYSTEM MEMORY MAP

[dumbed-down version of the Commodore 64 Programmer's Reference Guide; use the table found in there - wf]

APPENDIX II INPUT/OUTPUT REGISTER MAP

[same here - wf]

APPENDIX III DESCRIPTION OF FILES ON THE RELEASE DISK

ASSEMBLER64 See Section 7. This is the actual assembler program which loads into low memory and assembles the files which were created by the EDITOR program. To load the assembler, type LOAD"ASSEMBLER64",8 or use the wedge load command; then type RUN. Any source text files not previously saved will be lost since the assembler loads into the same area used.

BOOT ALL This program loads and starts the DOS WEDGE, the HI-LOADER, and the EDITOR all at the same time. These three programs reside in different areas in memory, allowing their use without having to reload before switching programs.

CROSSREF64 This program is used to print out the cross reference listing created by the assembler when that option is specified at assembly time. The program loads into low memory by using the LOAD"CROSSREF64",8 command and is started by typing RUN.

DOS5.1 This file contains the machine code for the wedge program. It is loaded automatically by running the DOS WEDGE64 or BOOT ALL program.

DOS WEDGE64 See Section 5. This program is the "boot loader" for the DOS 5.1 wedge program. It is the first program on the disk so that the LOAD"*",8 command can be used. After the program is loaded, type RUN and the wedge will be loaded and activated.

EDITOR64 See Section 6. This program is used to create and modify the source code files which will later be assembled. To load the editor, type LOAD"EDITOR64",8,1. After the program is loaded, type SYS49152 to activate. Then type NEW to clear the pointers before proceeding to create or edit any files. Be sure to save the source code file using th PUT command before loading the assembler.

LO-LOADER64 and HI-LOADER64 See Section 8. These two programs are used to laod the sequential records which are created by the assembler as its output or object file. When one of these programs is run, it loads the object file into memory in the specified location as true machine code which can be executed. The only difference between the two programs is that LOLOADER loads at hex $0800 and HILOADER64 loads at hex $C800. They are both included in case one of them interferes with the intended location for the machine code to reside.

To load the $0800 version, type LOAD"LOLOADER64",8 and then type RUN. To load the $C800 version, type LOAD "HILOADER64",8,1 and then type SYS51200.

MONITOR$8000 and MONITOR$C000 See Section 9. These two monitor programs are identical in function and are used primarily for load and save binary machine code files in their executable form.

They also allow the programmer many useful commands for examining and changing programs without having to run the whole assembly process. The first loads at hex $8000 by typing LOAD"MONITOR$8000",8,1 and is started by typing SYS32768. The second loads at hex $C000 by typing LOAD "MONITOR$C000",8,1 and is started by typing SYS49152.

APPENDIX IV 6500 SERIES MICROPROCESSOR INSTRUCTION SET OPCODES

ADC Add with Carry to Accumulator
AND "AND" to Accumulator
ASL Shift Left One Bit (Memory or Accumulator)
BCC Branch on Carry Clear
BCS Branch on Carry Set
BEQ Branch on Zero Result
BIT Test Bits in Memory with Accumulator
BMI Branch on Result Minus
BNE Branch on Result not Zero
BPL Branch on Result Plus
BRK Force an Interrupt or Break
BVC Branch on Overflow Clear
BVS Branch on Overflow Set
CLC Clear Carry Flag
CLD Clear Decimal Mode
CLI Clear Interrupt Disable Bit
CLV Clear Overflow Flag
CMP Compare Memory and Accumulator

CPX Compare Memory and Index X
CPY Compare Memory and Index Y
DEC Decrement Memory by One
DEX Decrement Index X by One
DEY Decrement Index Y by One
EOR Exclusive-Or Memory with Accumulator
INC Increment Memory by One
INX Increment X by One
INY Increment Y by One
JMP Jump to New Location
JSR Jump to New Location, Saving Return Address
LDA Transfer Memory to Accumulator
LDX Transfer Memory to Index X
LDY Transfer Memory to Index Y
LSR Shift One Bit Right (Memory or Accumulator)
NOP Do Nothing - No Operation
ORA "OR" Memory with Accumulator
PHA Push Accumulator on Stack
PHP Push Processor Status on Stack
PLA Pull Accumulator From Stack
PLP Pull Processor Status From Stack
ROL Rotate One Bit Left (Memory or Accumulator)
ROR Rotate One Bit Right (Memory or Accumulator)
RTI Return From Interrupt
RTS Return From Subroutine
SBC Subtract Memory and Carry From Accumulator
SEC Set Carry Flag
SED Set Decimal Mode
SEI Set Interrupt Disable Status
STA Store Accumulator in Memory
STX Store Index X in Memory
STY Store Index Y in Memory
TAX Transfer Accumulator to Index X
TAY Transfer Accumulator to Index Y
TSX Transfer Stack Pointer to Index X
TXA Transfer Index X to Accumulator
TXS Transfer Index X to Stack Pointer
TYA Transfer index Y to Accumulator

Appendix V A SAMPLE OUTPUT LISTING OF THE COMMODORE 64 ASSEMBLER

```
LINE#  LOC   CODE     LINE
00161  CCE1           ;
00162  CCE1           ; INIT THE MEMORY MANAGER (STARTUP COMES HERE)
00163  CCE1           ;
00164  CCE1           GOOO
00165  CCE1  A202      LDX #$02      ;MOVE THREE BYTES
00166  CCE3  BD DE CC  WEDGE  LDA JUMP,X
00167  CCE6  95 7C      STA CHRGOT+3,X
00168  CCE8  CA        DEX
00169  CCE9  10 F8      BPL WEDGE
00170  CCEB           ;
00171  CCEB  A5 BA       LDA FA       ;USE CURRENT FA FOR
```

```
                              DEVICE ADDRESS
00172  CCED  8D 77 CC        STA SVFA
00173  CCF0           ;
00174  CCF0           ; SAY HELLO AND EXIT
00175  CCF0           ;
00176  CCF0  4C 4B CF        JMP MSG       ;PRINT HELLO (JSR/RTS)
00177  CCF3           ;
00178  CCF3           ; THIS IS WHERE WE COME TO DO THE WORK
00179  CCF3           ;
00180  CCF3           START
00181  CCF3  85 A6           STA BUFPT     ;SAVE .A, .X
00182  CCF5  86 A7           STX BUFPT+1
00183  CCF7  BA             TSX           ;ACTIVATED CALL IN 'GONE'
00184  CCF8  BD 01 01         LDA $0101,X
00185  CCFB  C9 E6           CMP #<GONE    ;FROM A RUNNING PROGRAM??
00186  CCFD  F0 04           BEQ TRYTWO
00187  CCFF  C9 8C           CMP #<MAIN    ;FROM DIRECT MODE??
00188  CD01  D0 17           BNE NOTCMD
00189  CD03  BD 02 01  TRYTWO LDA $0102,X
00190  CD06  C9 A7           CMP #>GONE    ;PROGRAM?
00191  CD08  F0 04           BEQ FINDIT
00192  CD0A  C9 A4           CMP #>MAIN    ;DIRECT?
00193  CD0C  D0 0C           BNE NOTCMD
00194  CD0E  A5 A6    FINDIT LDA BUFPT     ;GET THE COMMAND BACK
00195  CD10  A2 08           LDA #NCMD-1
00196  CD12           FINDC               ;FIND THE COMMAND
00197  CD12  D0 19 CC        CMP CMD,X
00198  CD15  F0 11           BEQ CALL10
00199  CD17  CA             DEX
00200  CD18  10 F8           BPL FINDC
00201  CD1A           ;
00202  CD1A           NOTCMD
00203  CD1A  A5 A6           LDA BUFPT     ;RESTORE REGS
00204  CD1C  A6 A7           LDX BUFPT+1
00205  CD1E  C9 3A           CMP #':       ;COMPLETE CHRGOT
00206  CD20  B0 03           BCS STRTS
00207  CD22  4C 80 00        JMP CHRGOT+7
00208  CD25  4C 8A 00  STRRTS JMP CHRGOT+17  ;TO THE END OF CHRGOT
00209  CD28           ;
00210  CD28           CALL10
00211  CD28  86 A9           STX CNTDN     ;SAVE INDEX
00212  CD2A  8D 7A CC        STA FLAG      ;SAVE THE COMMAND FOR LATER
00213  CD2D  20 A3 CE        LSR RDFILE    ;GET FILENAME AND LENGTH
00214  CD30  A6 A5           LDX CNTDN     ;RESTORE INDEX
00215  CD32  A9 27           LDA #<FILE    ;SET FILENAME ADDRESS
```

APPENDIX VI EXPLANATION OF ERROR MESSAGES

Error messages are given in the program listing accompanying the statements in error. The following is a list of all error messages which might be produced during assembly.

**A MODE NOT ALLOWED

Following the legal opcode, and one or more spaces, is the letter A followed by one or more spaces. The assembler is trying to use the accumulator (A = accumulator mode) as the operand. However, the opcode int he statement is one which does not allow reference to the accumulator. Check for a statement labelled A (an illegal statement), which this statement is referencing. If you were tying to reference the accumulator, look upt he valid operands for the opcode used.

**A,X,Y,S,P RESERVED

A label on a statement is one of the five reserved names (A, X, Y, S and P). They have special meaning to the assembler and therefore cannot be used as labels. Use of one of these names will cause this error message to be printed. No code will be generated for the statement. The label does not get defined and will appear in the symbol table as an undefined variable. Reference to such a label elsewhere int he program will cause error messages to be printed as if the label were never declared.

**BRANCH OUT OF RANGE

All of the branch instructions (excluding the two jumps) are assembled into two bytes of code. One byte is for the opcode and the other for the address to branch to. The branch is taken relative to the address of the beginning of the next instruction. If the value of the byte is 0-127, the branch is forward; if the value is 128-255, the branch is backward. (A negative branch is in two's complement form). Therefore, a branch instruction can only branch forward 127 or backward 128 bytes relative to the beginning of the next instruction. If an attempt is made to branch further than these limits, this error message will be printed. To current, restucture the program.

**CAN'T EVAL EXPRESSION

In evaluating an expression, the assembler found a character it couldn't interpret as being part of a valid expression. This can happed if the field following an opcode contains special characters not valid within expressions (i.e. parentheses). check the operand field and make sure only valid special characters are within a field (between commas).

**DUPLICATE SYMBOL

The first field on the card is not an opcode so it is interpreted as a label. If the current line is the first line in which that symbol appears as a label (or on the left side of an equals sign), it is put into the symbol table and tagged as defined in that line. However, if the symbol has appeared as a labell, or on the left of an equate prior to the current line, the assembler finds the label already in the symbol table. The assembler does not allow redefinitions of symbols and will, in this case, print this error message.

**FILE EXISTS

The FILE EXISTS error message occurs when the object file named already exists on the diskette. This error can be corrected by scratching the old file or changing the diskette.

**FILE NOT FOUND

The FILE NOT FOUND error message is displayed when one of the following occurs:

o The source file was not found.
o A .LIB specifies a nonexistant file.
o A .FIL specifies a nonexistant file.

The user should make sure that the filename is not misspelled, or that the wrong diskette was placed in the disk drive.

**FORWARD REFERENCE

The expression on the right side of an equals sign contains a symbol that hasn't been defined previously. One of the operations of the assembler is to evaluate expressions or labels, and assign addresses or values to them. The assembler processes the input Source Code sequentially, which means that all of the symbols that are encountered fall into two classes, i.e., already-defined symbols and non-previously-encountered symbols. The assembler assigns defined values and builds a table of undefined symbols. When a previously defined symbol is discovered, it is substituted into the table. The assembler then processes all of the input statements a second time using currently defined values.

A label or expression which uses a yet undefined value is considered to be referenced forward to the to-be-defined value.

To allow for conformity of evaluating expressions, this assembler allows for one level of forward reference so that the following code is allowed.

```
Card Sequence   Label    Opcode   Operand
100                      BNE      NEWONE
200          NEWONE      LDA      #5
```

The following is not allowed:

```
Card Sequence   Label    Opcode   Operand
100                      BNE      NEWONE
200          NEWONE      INC      NEXT+5
300          NEXT        LDA      #5
```

This feature should not disturb the normal use of labels. The correction for this problem in this example is:

```
Card Sequence   Label    Opcode   Operand
100                      BNE      NEWONE
300          NEXT        LDA      #5
```

```
 301          NEWONE   INC     NEXT+5
```

This error may also mean that a value on the right side of the '=' is not defined at all in the program, in which case, the cure is the same as for undefined values.

The assembler cannot process more than one level of computed forward reference. All expressions with symbols that appear on the right side of any equal sign must refer only to previously defined symbols for the equate to be processed.

**ILLEGAL OPERAND TYPE

After finding an opcode that does not have an implied operand, the assembler passes the operand field (the next non-blank field following the opcode) and determines what type of operand it is (indexed, absolute, etc.). If the type of operand found is not valid for the opcode, this error message will be printed.

Check to see what types of operands are allowed for the opcode and make sure the form of the operand type is correct (see the section 1.1 on addressing modes).

Check for the operand field starting with a left parenthesis. If it is supposed to be an indirect operand, recheck the correct format for the two types available. If the format was wrong (missing right parenthesis or index register), this error will be printed. Also check for missing or wrong index registers in an indexed operand (form: expression, index register).

**IMPROPER OPCODE

The assembler searches a line until it finds the first non-blank character string. If this string is not one of the 56 valid opcodes, it assumes it is a label and places it in the symbol table. It then continues parsing for the next non-blank character string. If none are found, the next line will be read in and the assembly will continue. However, if a second field is found, it is assumed to be an opcode (since only one label is allowed per line). If this character string is not a valid opcode, the error message is displayed.

This error can occur if opcodes are misspelled, in which case the assembler will interpret the opcode as a label (if no label appears on the card). It will then try to assemble the next field as the opcode. If there is another field, this error will be printed.

Check for a misspelled opcode or for more than one label on a line.

**INDEXED MUST BE X OR Y

After finding a valid opcode, the assembler looks for the operand. In this case, the first character in the operand field is a left parenthesis. The assembler interpretes the next field as an indirect addrses which, with the exception of the jump statement, must be

indexed by one of the index registers, X or Y. In the erroneous case, the character that the assembler was trying to interpret as an index register is not X or Y and this error message is printed.

Check for the operand field starting with a left parenthesis. If it is supposed to be an indirect operand, recheck the correct format for the two types available. If the format is wrong (missing right parenthesis or index register), this error will be printed. Also, check for missing or wrong index registers in an indexed operand (form: expression, index register).

**INDIRECT OUT OF RANGE

The assembler recognizes an indirect address byt he parentheses that surround it. If the field following an opcode has parentheses around it, the assembler will try to assemble it as an indirect address. If the operand field extends into absolute mode, i.e., larger than 255 (two bytes would be required to specify the address), this error will be printed.

This error will only occur if the operand field is in correct form (i.e., an index register following the address), and the address field is out of page zero. To correct this, the address field must refer to page zero memory. (The implied high order byte is 00.)

**INVALID ADDRESS

An address referenced in an instruction, or the addrses in one of the assembler directives (.BYTE, .DBYTE, .WORD), is invalid. In the case of an instruction, the operand that is generated by the assembler must be greater than or equal to zero, and less than or equal to $FFFF (2 bytes long). (This excludes relative branches which are limited to +127 or -128 from the next instruction.) If the operand generates more than two bytes of code or is less than zero, this error message will be printed. For the .BYTE directive, each operand is limited to one byte. All address references must be greater than or equal to zero.

This validity is checked after the operand is evaluated. Check for values of symbols used in the operand field (see the symbol table for this information).

**LABEL START NEED A-Z

The first non-blank field is not a valid opcode. Therefore, the assembler tried to interpret it as a label. However, the first character of the field does not begin with an alphabetic character and the error message is printed.

Check for an unlabelled statement with only an operand field that does not start with a special character. Also check for an illegal label in the instruction.

**LABEL TOO LONG

All symbols are limited to six characters in length. When parsing, the assembler looks for one of the separating characters (usually a blank) to find the end of a label or string. If other than one of these separators is used, the error message will be printed providing that the illegal separator causes the symbol to extend beyond six characters in length. Check for no spacing between labels and opcodes. Also, check for a comment card with a long first word that doesn't begin with a semicolon. In this case the assembler is trying to interpret part of the comment as a label.

**NON-ALPHANUMERIC

Labels are made up of one to six alphanumeric digits. The label field must be separated from the opcode field by one or more blanks. If a special character or other separator is between the label and the opcode, this error message might be printed.

Each of the 56 valid opcodes are made up of three alphabetic characters. They must be separated from the operand field (if one is necessary) by one or more blanks. If the opcode ends with a special character (such as a comma), this error message will be printed.

In the case of a lone label or an opcode that needs no operand, they can be followed directly by a semicolon to denote the rest of the card as a comment (use of a semicolon tabs the comment out to the next tab position).

**PC NEGATIVE--RESET 0

An assembled prorgam is loaded into core in the range of position 0 to 64K (65535). This is the extent of the machine. A maximum of two bytes can be used to define an address. Because there is no such thing as negative memory, an attempt to reference a negative position will cause this error and the program counter (or pointer to the current memory location) to be reset to zero.

When this error occurs, the assembler countinues assembling the code with the new value of the program counter. This could cause multiple bytes to be assembled into the same locations. Therefore, care should be taken to keep the program counter within the proper limits.

**RAN OFF END OF CARD

This error message will occur if the assembler is looking for a needed field and runs off the end of the card (or line image) before the field is found. The following should be checked for: a valid opcode field without an operand field on the same card; an opcode that was thought to take an implied operand, which in fact needed an operand; an ASCII string that is missing the closing quote (make sure any embedded quotes are doubled; to have a quote at the end of the string, there must be three quotes, two for the embedded quote and one to close off the string); a comma at the end of the operand field indicates there are more operands to come; if there aren't other operands, the assembler will run off the current line looking for them.

**READ ERROR

This message refers to a disk drive read error. Refer to your disk drive manual for a description of these errors and their causes.

**UNDEFINED DIRECTIVE

All assembler directives begin with a period. If a period is the first character in a non-blank field, the assembler interprets the following character string as a directive. If the character string that follows is not a valid assembler directive, this error message will be printed.

Check for a misspelled directive or a period at the beginning of a field that is not a directive.

**UNDEFINED SYMBOL

This error is generated by the second pass. If in the furst pass the assembler finds a symbol in the operand field (the field following the opcode or an equals sign) that has not been defined yet, the assembler puts the symbol into the table and flags it for interpretation by pass two. If the symbol is defined (shows up on the left of an equate or as the first non-blank field in a statement), pass one will define it and enter it in the symbol table. Therefore, a symbol in an operand field, found before the definition, will be defined with a value when pass two assembles it. In this case, the assembly process can be completed. This is what is meant by one level of forward reference (See Forward Reference Error).

However, if pass one doesn't find the symbol as a label or on the left of an equate, the assembler never enters it in the symbol table as a defined symbol. When pass two tries to interpret the operand field the symbol is in, there is no corresponding value for the symbol and the field cannot be interpreted. Therefore, the error message is printed with no value for the operand.

This error will also occur if a reserved symbol A, X, Y, S, or P, is used as a label and referred to elsewhere in the program. On the statement that references the reserved symbol, the assembler sees it as a symbol that has not been defined. Check for use of reserved symbols, misspelled labels or missing labels to correct this error.

Note: When the assembler finds an expression (whether it is in an OPERAND field or on the right of an equals sign) it tries to evaluate the expression. If there is a symbol within the expression that hasn't been defined yet, the assembler will flag it as a forward reference and wait to evaluate it in the second pass. If the expression is on the right side of an equal sign, the forward reference is a severe error and will be flagged as such. However, if the expression is in an OPERAND field of a valid OPCODE, the first pass will set aside two bytes for the value of the expression and flag it as a forward reference. When the second pass fills in the value of the expression, and the value of the expression is one byte long i.e., 256, the instruction is one byte

longer than required. This is because the forward reference to page zero memory wastes one byte of memory (the extra one that was saved). During the first pass, the assembler didn't know how large the value was, so it saved for the largest value which was two bytes.

APPENDIX VII EDITOR64 COMMAND SUMMARY

Command	Description
AUTO n1	Starts automatic line numbering
AUTO	Shuts off auto
CHANGE/s1/s2/,n1-n2	Change string in line range
CHANGE/s1/s2/	Change string in entire file
CPUT"file"	Compacted PUT, unnecessary spaces are removed
DELETE n1-n2	Delete range
FIND/s1/,n1-n2	Find string in line range
FIND/s1/	Find string in entire file
FORMAT n1-n2	Print formatted
GET"file",n1-n2,n3	Bring in text from disk file
GET"file"	Short form GET
KILL	Disable the editor
LIST	List lines of text
NUMBER n1,n2,n3	Renumber text
PUT"file",n1-n2,n3,n4	Save text on disk drive
PUT"file"	Save text, short form

APPENDIX VIII MONITOR COMMAND SUMMARY

ASSEMBLE	A	Assemble a line of machine code
COMPARE	C	Compare two sections of memory and report differences
DISASSEMBLE	D	Disassemble a line of 6502 code
FILL	F	Fill memory with the specified byte
GO	G	Start execution at the specified address
HUNT	H	Hunt through memory for all occurrences of certain bytes
INTERROGATE	I	Show ASCII values of memory locations
LOAD	L	Load a file from tape or disk
MEMORY	M	Display the hexadecimal values of memory locations
NEW LOCATOR	N	Adjust machine language program after moving it
REGISTERS	R	Display the CPU registers
SAVE	S	Save to tape or disk
TRANSFER	T	Transfer code from one section of memory to another
EXIT	X	Exit 64MON (BASIC will need to be reset)

DOS WEDGE COMMAND SUMMARY

Command	Description
@	Current disk status
@C(dr):newfile([vol])=oldfile([vol])	Copy a file
@I(dr)	Initialize a drive
@N(dr):diskname,id	Format a disk
@Q	Kill the wedge program
@R(dr):newfile([vol])=oldfile([vol])	Rename a file

@S(dr):filename(*)([vol]) Scratch a file
@UJ Reset DOS
@$(dr):(filename)(*)([vol]) Read the directory
#n Direct DOS commands to device n
/filename Load a file (at BASIC)
%filename Load a file (at load address)
^filename Load a file (at BASIC) and run
<-filename Save a file
[single-character left arrow - wf]

Note: 'Vol' is any character enclosed in square brackers; 'dr' must be 0 (zero) or 1 (one) for the respective drives of a 2-disk drive.