ASSEMBLER DEVELOPMENT PROGRAM

ALL YOUR COMMODORE COMPUTER NEEDS.....

EDITOR featuring AUTO line number — RE-NUMBER which may be used for BASIC as well — PUT and GET for file management — PUTC and GETC for cassette — DELETE, FIND and FIND/CHANGE for source files as well as BASIC programs — plus a super text SCROLL system with automatic screen rewrite and line insert system.

ASSEMBLER will read source from memory, a disk file or a series of chained files — output to memory, hex file or loadable program file - conditional assembly - two pass efficient symbolic assembly - beautifully formatted and paged printed listings — comprehensive error trapping.

MONITOR aids debugging and investigative work — includes its own mini-assembler and dis-assembler — MEMORY SAVE and LOAD to disk or cassette — memory DISPLAY/MODIFY — hunt facility to find strings — fill memory — error/command channel access to disk.

DOS-SUPPORT helps with disk management — directory display — disk formatting and initialising — LOAD and LOAD+RUN — binary load with optional off-set.

note: details vary between versions depending upon facilities built into each machine.

16	plus/4	64	128	128D
	✓	✓	✓	✓

0 43467 02870 5

WILL RUN ON 1541, 1570, 1571, 1551

commodore

MULTISOFT

Assembler

and

Program Development System

FOR

C128 C64 and PLUS/4 Computers

A comprehensive package for all levels of machine language application and BASIC programming.

commodore

```
**************************************
*         JCL SOFTWARE LTD           *
*                                    *
*    6500 SERIES ASSEMBLER. V3.00    *
*                                    *
*  FOR  THE  COMMODORE  64 COMPUTER  *
*                                    *
*    By Richard Leman C.Eng MIERE    *
*                                    *
**************************************
```


DISCLAIMER

A considerable amount of care has gone into the development and testing of this product and it is beleived to be reliable and suitable for use. JCL Software Ltd or any agent acting on their behalf cannot accept responsibility for the direct or indirect consequences of the use of this product.

Rev 25.02.86

PREFACE

The JCL SOFTWARE ASSEMBLER DEVELOPMENT SYSTEM software package allows you to program in the native 6500 series Assembly language code, directly on your Commodore computer. It provides you with very powerful Assembler, Editor, and program development aids. Some of the development aids may also be used when writing BASIC programs. 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 this user's manual is directed towards the experienced computer user who already has some familiarity with the 6500 series Assembly language and the operation of his Commodore computer. It is not intended to provide the knowledge of "how to" in assembly language, but provides the software tools for the experienced assembly language programmer.

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

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)

In addition, the Programmer's Reference Guide specific to your computer is a most valuable reference book.

TABLE OF CONTENTS

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 6500 series processor 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 specified by the user.

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

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

NAME
Generally, NAMES specified in UPPER CASE indicate the actual input to be typed.

BASIC key words, program lines, assembler commands and mnemonics are generally shown in upper case for clarity of presentation. When entering such material into a computer the screen appearance may be different if the lower case mode is selected, but the function acheived will not normally be changed.

IMPORTANT

This manual describes the Editor, Assembler and other features of the package for C128, C64 and Plus 4 computers. The facilities provided by each version depend on the architecture of each machine and the built in facilities. For example the C128 has an excellent memory monitor so the program does not include one. The C64 does not have a monitor so the program provides the needed facilities. Where necessary, machine specific facilities are identified by the following symbology :- (*64 ONLY*) or (*NOT 64*)

INSTALLATION

1. CARTRIDGE EDITORS AND ASSEMBLERS
Some models of the JCL Software Assembler package include a cartridge containing part or all of the software. In such cases the cartridge must be plugged in before the package will work.

IMPORTANT

*** BEFORE INSTALLING OR REMOVING THE CARTRIDGE ***
********** SWITCH THE COMPUTER OFF **********

Fit the cartridge with the label side upper-most (or facing forwards for the SX-64) engaging the unit in the socket in the cartridge aperture, and carefully pushing until full engagement is acheived. Switch the computer ON in the normal manner, and when the screen display appears type the SYS call that appears on the label. For example, the C-64 requires :-

SYS 32768 and press RETURN

The computer will respond with a list of the new words that you will be using to write, assemble and run machine code programs. You can see this list at any time by typing :

WORDS and pressing RETURN

2. CARTRIDGE EDITOR WITH SOFT LOADED ASSEMBLERS
Packages consisting of a cartridge and a soft loaded assembler should first have their cartridges installed and activated as described above and then the assembler should be loaded from disk. This is most easily done by typing the ASM command which will re-prompt with the appropriate command to load the assembler from disk. For example the ASM command on the C-64 version will reply with :

%HS64.MOD

Place the assembler disk in drive 0 and press the RETURN key. After the assembler has been loaded further use of the ASM command will cause the assembler to run.

3. COMPLETE SOFT LOADED PACKAGES
Soft loaded version of the package always includes a loader which is placed first on the disk so it may be loaded with LOAD "*",8. RUN the loader and it will locate the rest of the software into the appropriate places in your computer.

INTRODUCTION

This manual describes the Assembly Language and assembly process for Commodore computers 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 and the processor used in the C128 (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 machine form as object code. The symbols used and rules of 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 micro-computer system has fewer resources than are needed for an assembler. However, in the case of a Commodore computer operating with the JCL Software Assembler this is not true. With a floppy disk and printer, the system is very well suited for software development. When the assembler is supplied in cartridge form it may be used for the development of small programs without the use of floppy disks, a standard cassette drive providing the minimum facilities needed to save and load source files and save the finished program.

Normally, digital computers use the binary number system for representation of data and instructions. Computers understand only ones and zeroes 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 move 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 listed 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 source 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 many 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), addresssing 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.

1.0 INSTRUCTION FORMAT CONVENTIONS

Assembler instructions for the JCL Software 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 and subtract). Expressions are evaluated by the assembler to compute 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 non-blank 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 VALUE FROM BETA 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 computed 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 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 01 3C 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 directly into 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
'x'	(Apostrophes) specifies an ASCII literal character in immediate instructions

The absence of a prefix symbol indicates decimal 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 the character G into the accumulator. Since the accumulator is one byte, the value loaded must be in the range of 0 to 255 decimal.

1.3 Relative

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

```
LDA VAL1    ;GET 1ST VALUE
CMP VAL2    ;COMPARE
BEQ START   ;IF EQUAL BRANCH TO START
```

if the values compared are equal, a transfer to the instruction labelled START is made. The branch address is a one byte positive or negative offset which is added to the program counter during exection. 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.

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. Their assembly language formats are as follows :

```
ASL A
LSR A
ROL A
ROR A
```

1.5 Indexed Indirect

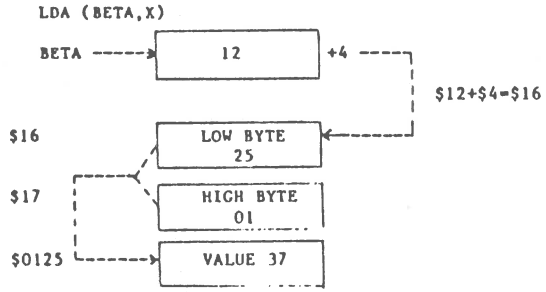
In this mode, the operand address is computed by first adding the X registed (the index) to the argument in the operand (in the example below, BETA). The resulting value is the indirect page zero address which contains the actual operand address. In the example:

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

the parentheses around the operand indicates indirect mode. In the above example, the value in index register X is added to BETA. 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 $4
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 of address vectors in page zero.) This form of addressing is shown in the following illustration.



1.6 Indirect Indexed

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

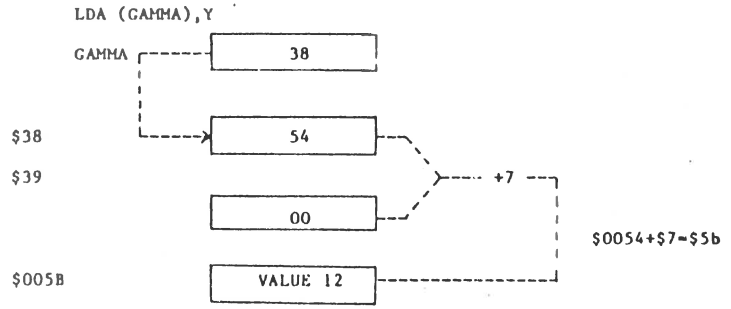
```
LDA  (GAMMA),Y
```

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 $7
Locations $0039 and $0038 contain $00 and $54
Location $005B contains $12
```

The address at $38 is $0054; seven is then added to this, giving an effective address $005B. The value at $005B is $12 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.



2.0 ASSEMBLER DIRECTIVES AND CONDITIONAL ASSEMBLY

2.1 Directives

There are ten assembler directives used to reserve storage and direct information to the assembler. Eight have symbolic names with a period as the first character. The ninth, a symbolic equate, uses an equals sign (=) to establish a value for a symbol. The tenth, 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 *=$200. A list of the directives is given below and their use is explained in this section.

```
.BYTE   .WORD   .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 contain 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.)

```
HERE   .BYTE 2,<TABLE1,<TABLE2
THERE  .BYTE 1,$0F,@03,%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.

Arithmetic operations in the .BYTE directive are supported in this package provided that they conform to the rules that apply to the solution of arithmetic operations in the operand field.

.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 addresses 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 values in this order.

The hexadecimal portion of the example ($FF03) would be stored 03,FF.

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 refer to, or 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 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   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   5       skip five lines
```

.OPT is used to control the generation of output fields and listings. The options available are:

```
LIST, NOLIST;    SYMBOL, NOSYMBOL;    WIDE, NARROW
```

Default settings are:

```
.OPT    LIST
.OPT    SYMBOLS
.OPT    WIDE
```

Here are descriptions for each of the options:

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

SYMBOL/NOSYMBOL:
Controls the printing of the symbol table on completion of assembly. The table is only printed if the ',p' (print) option is exercised when replying to the assembler prompt line.

WIDE/NARROW:
The assembler will limit the number of characters to each printed line to 80 if the NARrow option is selected, else the full line length will be output. This facility can be useful when 80 column printers are 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 statement. 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 reference to a library file or contain a .FIL. A '.FIL' terminates assembly of the file containing it and transfers source reading to the file named on the OPERAND. These 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 .END directive.

2.2 Conditional Assembly

Conditional assembly is means of exercising control over the assembly process to determine which parts of a source code are to be assembled.

Conditional assembly involves the testing of a value or expression and then assembling the next section of source code depending on the result of the test.

Two options are provided. The first evaluates an expression and assembles the following code if the result is zero :

```
.IFE (expression) <

    Code here is assembled if the
    result of the expression is
    equal to zero

>
 Un-conditional assembly re-commences here
```

The second option assembles the following code if the result of evaluating the expression is NOT zero :

```
.IFN (expression) <

    Code here is assembled if the
    result of the expression is
    NOT equal to zero

>
Un-conditional assembly recommences here.
```

These important rules MUST be followed :

1.The dot of the .IFE or .IFN conditional assembly directives must be the first non-blank character on the source line.

2.The expression must contain numeric values or labels that have already been defined, and must not include spaces within the expression. The expression should terminate with a space and a < symbol.

3.The > symbol which marks the end of the conditionally assembled code must be the first character on the line.

4.Conditional directives cannot be nested.

Examples :
Consider the development of an assembler. Several versions are to be produced from the same source code. One for the Commodore 64 and another for the Commodore B Series, or 700 Series machines. Obviously it is necessary to make various declarations about the memory map of each machine, and throughout the program various machine specific pieces of code need to be written, although the majority of the code is common to both versions. It is too time consuming to keep two sets of source files on separate disks, one for each machine, and

this would also mean that when a change is made on one version an equivalent change would need to be made on the other - twice the chance of making an error !!

Using the conditional assembly facilities the problem becomes simpler. Two short files are prepared called HS64 and HS700. These set a flag called COMPUT to 64 in one case, and 700 in the other. Both chain into the first source file using .FIL. Thus assembling HS64 produces the C-64 program and assembling HS700 produces the B Series equivalent.

Throughout the many files that comprised the entire source code the value of COMPUT is tested and sections of code assembled or disregarded as appropriate.

This illustration shows how one or other of two definitions files were read.

```
;READ DEFINITIONS FILE
.IFE COMPUT-700 <
.LIB DEFINE700
>
.IFE COMPUT-64 <
.LIB DEFINE64
>
```

By setting a flag called DEBUG various values may be shown on the screen. Once the program was working correctly DEBUG is set to zero and the next assembly pass ommitted the debugging code.

```
.IFN DEBUG <
       JSR BSOUT  ;PUT RESULT ON SCREEN
>
```

Conditional expressions can include more than one reference :

```
.IFE COMPUT+DEBUG-65 <
       JSR OUTDSP  ;SHOW CONTENT OF BUFFER
>
```

This last case only called OUTDSP when the C-64 version was being assembled and DEBUG was true. The B Series version never assembled the call.

3.0 HINTS AND TIPS

The following suggestions are offered with the caveate that rules are for guidance, and sometimes for breaking! They are a distillation of a number of years experience programming on Commodore Computers, for better or worse!

1. Design each program on paper first, unless it is a trivial item. Consider memory mapping, I/O devices and key functions carefully.

2. Try to use a top-down approach. Design your software to call major operations as subroutines so you can write simple dummy routines, or stubs, to stand in for undeveloped areas of code. The overall structure may then be tested before the whole program is completed.

3. Ten minutes thinking before trying to fix a problem is better than several hours of debugging a bad patch.

4. If the going gets tough, take a break. In my experience a problem that is not solved by nine p.m is unlikely to be fixed by midnight - and gives itself up in ten minutes the next morning.

5. Group all definitions for a project into a file. Similarly, make a definition file for each machine you work with defining zero page, porting, kernal etc.

6. Try not to re-invent the wheel. Group popular subroutines together in a single file and call them with .LIB for each new project.

7. Be expansive with space for variable storage. Re-use of flag locations etc is a good idea when memory is tight but can lead to clashes if the program starts to expand. Group all variables together and initialise them all with a block fill routine before setting up special values etc.

8. The bug is most likely to be in the last thing you changed.

9. Document - document - document. A modification or extension needed in a hurry six months after the program was finished (?) goes through a lot more easily if you can read about each bit of the program before changing it. Writing it down clears the mind wonderfully.

10. Describe each procedure in words and then in a real or pseudo high level form before writing the machine code. Keep the line by line level comments simple and short.

11. Stare at a piece of malfunctioning source code for only as long as it takes you to be sure that it is correct - twice. Then say to yourself - the bug is somewhere else, probably in a subroutine which corrupts a common variable or which destroys a flag.

12. Talking a problem through with another programmer can be a useful way of homing in on a problem area. Nine times out of ten he will not need to say anything!

13. Backup - backup - backup. Use the DO facility in the editor to keep your files under their proper names. Copy all files to another disk and keep them in another building. If ever you suspect a disk corruption make a duplicate before trying any recovery actions.

4.0 OUTPUT FILES GENERATED BY THE ASSEMBLER

There are four output files generated by the assembler. Each file is optional and can be created through the use of assembler directives or by exercising one or more of the available options when responding to the assembler prompt. The LISTING file contains the program list with errors and the symbol table. By suitable choice of assembly option and directive the listing file can be reduced to a list of errors only, the full listing being produced when all errors have been eliminated. The INTERFACE file contains the object code for the loader programs used by the earlier CBM assemblers and also by the JCL Software EPROM programmers. The PROGRAM file is a directly loadable file which may be executed after being loaded into memory. Finally, the SYMBOL file is a list of the labels and their assigned values produced in a form that may be loaded by the editor.

Listing File and Error File

The listing file will only be produced if the ',s' or ',p' (screen or printer) option is specified when the source filename is given. This file is made up of two sections: Program and Error List, and Symbol Table.

This listing contains the source statements of the program along with the assembled code. Errors and warnings 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. If a NOLIST directive is placed in the source code only the error lines and warnings will be produced. Error lines and warnings are always displayed on the screen.

The symbol table will be produced if the ',p' option is exercised when the source file name is given unless the NOSYM option is used. It contains a list of all symbols used in the program sorted into label order, and their addresses or values.

Interface File

This file does not contain true object code, but data which can be loaded and converted to machine code by a loader, or read by an EPROM programmer. 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 thru 12, etc. The 8-bit byte 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.

Program File

The program file is generally produced when all assembly errors have been corrected by one or more assembly cycles and the program is ready for testing. It is produced during assembly if the ',m' option is chosen and is written to the project disk using a filename consisting of up to twelve characters (or less) of the source file name plus the suffix .MOD. The program file may be loaded into memory by various means, and carries a load address specified by the first byte of assembled code.

Symbol file

The assembler creates a table containing each label used in the source code together with the value or address assigned to it. On completion of assembly the table is sorted into label order and used to print the symbol table, or written to disk as the symbol file under a name consisting of the first twelve characters (or less) of the source file name plus the suffix .SYM. The symbol file will be created if the directive .SAVesymbols is placed in the source listing at any point. The symbol file may be used to determine the position of a label for debugging purposes, or for declaring the position of all labels to another program module under development.

5.0 ADDITIONAL BASIC COMMANDS.(DOS SUPPORT) ** C64 ONLY **

The C64 version of the Assembler includes the popular DOS Support system which will aid you in performing disk housekeeping functions (copying, scratching, renaming), reading the directory, initializing 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 ACTIVATING THE DOS SUPPORT SYSTEM

The DOS Support system is automatically turned on when the assembler system is first activated. No further action is required by the user. Most of the commands may be used by programmers writing BASIC as well as machine language.

5.2 USING DOS SUPPORT

DOS Support provides 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), commands to load and run programs and the capability to perform operations using a wild card filename (any file whose name begins with certain characters).

Each command begins with a single character. (See 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. The ↑ (up arrow) is used to begin the command to load (at BASIC's Start of Text address) and automatically runs a program. The / (backslash) 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.

5.3 DOS SUPPORT COMMANDS

A description of each command is given in the following pages.
Appendix IX provides a brief summary of the 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 CLOSE15
```

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

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.

@N(drive):diskname,id

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

@R(drive):newfile=oldfile

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

@C(drive):newfile=(drive):oldfile

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

@S(drive):filename(*)

This command scratches the file specified by filename. If * is specified, all files beginning with the letters specified by filename will be scratched.

@UJ

This command will reset the DOS. Note that repeated use of this command may "hang" a 1541 disk drive requiring it to be switched off and on again.

@I(drive)

This command will initialize the disk drive.

/filename

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

```
/CALC-10
```

will cause the program named "CALC-10" to be loaded into memory. This command does the same thing as the BASIC command:

```
LOAD "CALC-10",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 "filename",8,1
```

where filename is the name of the program to load.

It is sometimes useful to be able to load a binary file into memory at an address other the address specified in the file header. This may be acheived with the form :

```
%filename,(addr)
```

The load address must be specified as a decimal value in the range 0 to 65535. Variables may not be used and will result in a SYNTAX ERROR.

^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.

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, DO, ASCII, CBM, TYPE and SCROLL. All of the commands are detailed in 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 files is in naming. Due to the method in which the assembler parses, spaces are not allowed in filenames. The files are sequential and when listing a directory will show as file type SEQ.

6.1 Activating the Editor.

The EDIT mode of operation is selected by typing the word "EDIT" and pressing RETURN. Edit mode is automatically selected if a GET "filename" command is used to load a source file. Note that edit mode is cancelled by typing the command "BASIC" or when a BASIC program is loaded with the / or ^ DOS Support commands.

6.2 Using the Editor

When the Editor 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.

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, insure 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.

Refer to Appendix VII for an Editor Command Summary.

6.3 Editor Commands

AUTO Line Numbering * C64 ONLY * (C128 and Plus4 - standard feature.)

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

```
AUTO nl
```

where nl is the increment between line numbers printed. To disable the AUTO function press RETURN when the cursor is immediatly against a new line number and on a line which is otherwise empty or type the AUTO command without an increment.

FIND string

The FIND command is used to search for and locate specific character strings in text. Each occurrence of the string is displayed on the screen. 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)

The FIND function may be used when programming in BASIC. In this mode the delimiter character may be chosen to be a quote mark in which case only strings will be inspected for a match.

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. (Optional) The format is the same as the LIST command in BASIC. If omitted, the whole text is searched.

The CHANGE function may be used in BASIC mode and handles tokenised material correctly.

DEL (Delete) ^ C64 Only ^ (C128 and Plus4 - standard feature.)

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

DEL n1-n2

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

TYPE

The TYPE command will print source code held in memory or in a file, either to the screen or to a printer. This is a convenient way to inspect a file without loading it into memory or of producing a printed copy of a file. A TYPE operation may be terminated with the RUN/STOP key or slowed down with the CTRL key in the same manner as LIST.

Printed output produced by TYPE will normally conform to CBM ASCII code but

may be switched to standard ASCII code by typing the command "ASCII" and pressing RETURN. (Reset to CBM style of output with "CBM".)
Page length, margin size and head+foot space default to 68 lines per page, an indent of zero characters and a head+foot totaling 8 lines respectively but may be changed to other values by adding additional parameters to the "TYPE" command.

Examples :

TYPE	Display current text on screen.
TYPE "filename"	Display a source file on the screen.
TYPE u4	Print current text held in memory.
TYPE "filaname",u4	Print a source file. (NB Device 4 only.)

Adding any, or all of the following parameters will adjust the form of the output to the screen or printer. The order of specification is not important.

,m xx	use a margin of xx spaces.
,l yy	print 'yy' lines per page.
,h zz	print 'zz' blank lines for 'head+foot'.

GET Files

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

GET "filename" ,(n1)

n1	Begins inputting source at this line in the file currently in memory (Optional)

GET C "filename" ,(n1)

This special form of GET will load a file from a cassette drive.

Note: GET starts numbering lines at 1000 and increments 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 text in memory.

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)

n1	Starting line number (Optional)
n2	Ending line number (Optional)

If n1-n2 are left out, the whole file is written to the disk.

PUT C "filename"

This special form of PUT will save the current source text on a cassette drive.

DO

This command executes the first line of a source file or a BASIC program as if it had been typed from the keyboard, provided that it starts with a semi-colon (;) or a REM statement. It is good practice to include the correct name of a file as the first line. The following examples show how this can be turned to better account by providing a convenient way of replacing a file on disk :

```
BASIC     10 REM SAVE"@0:filename",8 : VERIFY"0:filename",8

SOURCE
FILE      1000 ; PUT"@0:filename"
```

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 (Commodore 64)

The RENumber function allows the user to renumber all or part of the file in memory.

```
REN       (n1),(n2),(n3)

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

RENumber may be used in BASIC programming as well. GOSUBxx, GOTOxx, ONxx,yy and RUNxx statements are correctly adjusted.

If the parameters n1, n2 and n3 are not specified then default values are inserted. BASIC programs will be renumbered in steps of 10, with the first line being set to line 10. Assembler source material will be renumbered in steps of 10, with the first line being set to 1000.

NUM *C128 and Plus4*
The C128 and Plus4 have the RENUMBER function as standard for BASIC. The NUM command is used solely to generate new line numbers for ASCII source files. The parameters should be specified in the same manner as the C64 REN, and defaults are selected in the same manner.

6.4 General purpose Editor functions

JOIN "(drive:)filename"

JOIN provides a means of concatenating a BASIC program file to the end of a BASIC program held in memory. The source device is assumed to be the disk drive (device 8) unless the special form JOIN C is used, in which case the program will be read from cassette.

When using JOIN it is important that line numbers of ´joined´ modules are in correct order. If a module with low line numbers is joined to the end of a program with higher line numbers there is no simple way of correcting the situation.

SCROLL

This command activates several invaluable screen listing aids.

First, whenever the cursor reaches the top or bottom of the screen further text lines are ´scrolled´ on to the screen. If the cursor direction is reversed so that it moves away from the screen limits normal operation occurs and text on the screen may be modified with the normal screen editor.

The next three commands relate to clearing the screen and listing a section of text. (Function key numbers in brackets refer to the Plus4.)

Press F1 to re-list the screen from the first line number found by scanning from the top of the screen downwards.

Press F3 (F2) to re-list the screen from the line on which the cursor is located. This provision is very helpful after FIND has located several occurences of a wanted expression. The cursor is run up to the listing of the occurence most probably wanted and f3 will show this line and several following lines.

Press F5 (F3) to see the next ´screen´ of the program. The last line number on the screen is located, and the screen filled with listing from this point.

The F7 key (*C-64 only*) provides access to several functions, only when editting a BASIC or assembler source line. Press F7 and then a further key from the following list :

```
F7  E  Clear from cursor to END of line.
F7  S  Clear from cursor to START of line.
F7  T  Clear screen to TOP, starting at the cursor line.
F7  F  Clear screen to FOOT, starting at the cursor line.
F7  I  INSERT a blank line by scrolling the screen down.
       (If the line above the cursor starts with a line number
       then a line number one greater is automatically written;
       provided that a line with this number does not exist.
       Use SHIFT I if the line number is not wanted.)
F7  X  Return to normal cursor; no action.
```

CBM

Selects standard CBM code for printed output generated by the assembler and by TYPE.*

ASCII

Selects ASCII code for printed output generated by the assembler and by TYPE for users with ASCII printers interfaced to their computers.

RBAS (*C64 Only*)

The Reset BASic function resets the pointers that identify the start of the BASIC text space and execute a CLR. It is provided to simplify the process of saving hybrid programs (BASIC and machine code) on the C-64, made using the method described in Section 8.2.

SIZE filename

A program file has a natural load address which is specified by the first two bytes of the file. To determine the load address and also the address of the last byte of the program once loaded use the SIZE function.

```
SIZE "filename"

$start-$end
```

The start and end address' shown are in hexadecimal.

SETBRK and CLRBRK (*C64 Only*)

These two commands are used to insert and remove break points from a program being tested in RAM. The SETBRK command must be followed by a FOUR digit address in hexadecimal. (For example : SETBRK 3CB0) A BRK instruction is inserted into RAM at the address specified, and the original content and address preserved in memory. The byte preserved is also echoed on the screen for information.

If SETBRK is called when a breakpoint has already been set then a call to CLRBRK is made first.

CLRBRK reads the address and byte value saved by SETBRK and restores the program to its original form.

HD and DH

These commands convert a value from hexadecimal to decimal, and decimal to hexadecimal respectively. HD must always be followed by a FOUR digit hexadecimal string.
DH will accept values in the range 0 - 65535 ($0 to $FFFF), but does not include over range testing.

BOOM

Calls the machine cold start vector.

EXPAND (*C128 Only*)

It is good practice to enhance the readability of source files by indenting mnemonic, operand and comment fields. This practice results in a lot of wasted space in each file and slower operation of the assembler. (Despite this, the author finds a tidy layout makes changes and subsequent updates so much easier that the wasted space is a small penalty to pay!)

When the EXPAND function is switched ON source lines up to a comment are compressed as they are entered to remove un-necessary spaces. The SCROLL functions, FIND and the screen re-write functions expand source lines to a tabular form for readability.

Spaces within comments and .BYT 'xxx' expressions are not compressed.

Expand is switched ON and OFF by typing EXPAND and pressing RETURN.

When SCROLL is selected and EXPAND is ON, source lines typed in an un-tidy manner may be instantly displayed in tidy form by pressing F1, which re-writes the screen.

7.0 ASSEMBLING A SOURCE FILE

Once a source file is ready to assemble, you should first save it by using the PUT command.
The assembler keeps the symbol table in RAM, and the 2000 (C128 -4000) entry limit requires the reservation of 16K (32K) of memory.

The maximum amount of source material that may be kept in memory when the assembler is used is 14K for the C-64 and 26K for the Plus4. If the assembler is called and the pointer to the top of text is above the symbol table start then the assembler will return to command level with a warning message :

*** SOURCE TEXT AT RISK ***

If this occurs it is essential that the source is saved first, and a NEW command issued before attempting to use the assembler again.

Source code may be assembled directly from memory, dispensing with the need to assemble from a disk file. The speed of operation in this mode is very high. Cassette users will find this feature invaluable because text errors can be eliminated before it becomes necessary to save the source to tape.

(S L O W L Y !)

Source text assembled from memory may include .LIB and .FIL references.

7.1 ACTIVATING THE ASSEMBLER

To use the assembler type "ASM" and press return. The assembler will print a copyright notice and the user prompt line.

7.2. USING THE ASSEMBLER

When a program is being assembled, the user has the option of creating three types of output file on disk.

The first type is an object file which contains the data necessary to create a machine code program, in the hex format that has been historically supported by Commodore Assemblers. These files may be read by loaders and in addition are the one of the standard input formats for the JCL Software EPROM programmer.

An alternative, and frequently more useful file is a directly loadable program (or binary) file. This file may be loaded into memory for immediate execution and carries a load address set by the source file line that first generated an output.

It should be noted that the assembler will not overwrite either of these files unless instructed to do so.

The third type of file, WHICH IS ALWAYS WRITTEN WITH REPLACE is a symbol table file, and is created when a .SAVE instruction is included in the source file. This file consists of the symbol table created during assembly together with the value asigned to each symbol. The symbol table file carries a suffix .SYM on its filename and is a sequential file. Each entry is of the form :

(label) = $(value)

This file may be inspected by loading into memory using GET, or used as a definitions file for later projects.

When the assembler is called with ASM a prompt line asks :

SOURCE,OPTS (dr:filename,s,p,o/m/x,c)
?

The required response is the name of the source file followed by one or more options. The source file name is also used to name any output files produced, so it becomes a simple matter to keep all files related to a single project correctly named. If output files are to be re-written because they already exist place an @ symbol at the start of the source file name.

If a drive number is specified then the assembler looks for all parts of the source file on the specified drive and also creates the output files on the same drive.

,m
Results in the production of a module file which is a program file suitable for immediate loading. The module file is produced carrying a name consisting of the source file name with the suffix .MOD.

,o
Causes the assembler to create a hex format object file. The sequential file produced in this manner carries a name consisting of the source file name with the suffix .OBJ.

,x
Results in the assembler writing directly to core so that the resulting code may be executed immediatly. Section 8.2 describes safe areas for code assembled using this facility.

Note that a hex file, a module file or a write to core can be produced on one assembly cycle.

,p
Will produce a printout. Note that if printing is allowed by using the ´,p´ option individual sections of the source may be listed or not listed by the inclusion of .LISt and .NOList commands in the source file. The default is to .LISt.

,s
Obeys the same rules as ´,p´ but output is directed to the screen. During each assembly cycle it is possible to toggle the screen listing flag to turn the listing on/off regardless of the initial state selected by the ´,s´ option.

,c
Normal operation of the assembler is to read the source file from disk, but for small projects selection of the ´,c´ option will cause the assembler to read the current source held in core.

After completing the response line press RETURN. To skip assembly, delete the response and press RETURN.

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 command level. Press the S key to toggle the screen listing flag. Pressing any other key will continue the assembly process. This feature is useful for users without printers, as parts of the listing can be examined during assembly.

COMMAND FILES

The assembler may be run with a SYS call which includes the filename and assembler options to be used. In addition, the assembler provides an error flag which is zero if the there were no assembly errors. With these facilities it becomes possible to write a BASIC command file that controls the entire assembly process.

A typical C64 command file would be run by the DOS Support LOAD and RUN symbol. eg:

```
^CMD
```

The command file would then carry out the following functions automatically:

1. Scratch unwanted files to provide the greatest amount of disk space for the assembler.

2. Assemble the source file/s.

3. Terminate if there was an error, or load the resulting binary file with LOAD "filename.MOD",8,1

4. Call the loaded program with a SYS call.

A typical command file including the appropriate calls for the each machine will be found on the system disk.

Important note: *C64 and Plus 4* command files should not asign any BASIC variables unless MEMTOP is lowered to an address below the symbol table.

8.0 LOADING A FINISHED PROGRAM.

8.1 Methods of loading machine code programs.

The JCL Software Assembler dispenses with the need for a loader to read ASCII files into memory as it produces a loadable program (or binary) file directly.

Machine code programs may be loaded into memory by means which depend on the machine in use.

Commodore 128 series machines provide the BLOAD function that will load a program file at any location within memory. Eg:-

```
BLOAD "filename",B15,D0,P1024
```

The C-64 and Plus4 machines utilise a secondary address system to flag that a file is not a BASIC program, and that it is to be loaded at the natural load address (determined by the first two bytes of the program file, lo, hi.) and also that re-linking is to be by-passed :

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

After a LOAD called from within a program is completed, BASIC will resume operation at the beginning of the program, but variables are not destroyed. A simple program statement such as this will load a utility suite:

```
10 GG=GG+1:IF GG=1 THEN LOAD "filename",8,1
```

During program debugging the DOS Support '%' function will load a program type of file into its natural load address, also without re-linking, etc :

```
%0:filename
```

8.2 Integrating BASIC and Machine code

The two most significant differences between machine language and BASIC are the speed of execution and the ease of programming / debugging. A program written in BASIC can usually be finished to specification in a much shorter time than the equivalent program in machine code. The speed of execution of the BASIC version may leave a lot to be desired. Most operations carried out by a machine code program can be carried out in BASIC, again provided time is not the most critical parameter. The speed of execution of machine code can be several hundred times that of BASIC and this is probably the main reason machine code programs are written.

A popular compromise is to write parts of a program in BASIC and the remainder in machine code, thus taking advantage of the good points of both systems. Such a program is called a HYBRID. A well designed hybrid program loads in the same manner as an ordinary program, and needs no special action on the part of the user.

In a typical Commodore system RAM is allocated for program storage, variable storage and system use. Buffer areas are popular locations for transitory machine code utilities but the safest areas to use are inside the program area, in a space protected from the operating system by the adjustment of the various system pointers.

Here is a brief description of the most popular methods of producing hybrid software.

Method 1
Lower the top of memory pointers to convince the operating system that is has less RAM than usual, and then load the machine language utilities in to the reserved space. The reference manual usually describes the zero page locations defining the top of memory (MEMTOP is the label generally used). Lower the top of memory before any variable are declared, and do a CLR to bring all variable pointers into agreement.

This method has the advantage of offering a fixed address for the machine code origin, and correct use of a jump table allows the BASIC program to access the machine code via stable SYS calls.

On the debit side, the top of memory is a popular place for debugging routines and utilities so a clash over memory space can arise.

Method 2
Write the BASIC section, and fix the machine code utilities just above it. Before saving the program the end of text pointers are raised to include both BASIC and machine code. On loading back into the machine the operating system cannot distinguish between the BASIC and machine code and sets the end of text pointer to include both sections.

There are no particular advantages to this method, and many disadvantages. First, the BASIC program cannot be modified without first saving the machine code section for re-joining. Problems can arise due to the real end of BASIC not being where the operating system thinks it is, but the worst feature is that a need to re-assemble can arise if the BASIC section becomes larger and overlaps the start of the machine code. In addition, the entry point table will move after re-assembly, and the BASIC SYS calls will need to be adjusted or computed by examining the appropriate BASIC pointers.

Method 3
At the normal start of BASIC place a single line with a SYS call that directs the interpreter to a short machine language program which resets the start of BASIC to a higher address where the main BASIC program is stored, thus reserving a space for machine code. The same routine also adjusts the BASIC TXTPTR to the start of this new BASIC area and then performs an RTS. The net result is that the main BASIC program then runs and can call the machine code in the lower section of memory using SYS calls.

Once this method is mastered it will be found to be the best way to integrate machine code into BASIC. The finished program loads and runs as if it were all BASIC. The BASIC section can be modified without worrying about clashes with the machine code. Also, the BASIC program can access the machine code through a jump table at a stable address at the start of the machine language section.

If the BASIC section is modified, all the programmer need do before re-saving the hybrid is reset the bottom of BASIC pointers in zero page. (TXTTAB).

This method of integrating machine code and BASIC was developed by JCL Software on the original PET 2001 machine and has worked well on all later CBM single bank machines. Because the method involves placing a few lines of code at either end of the main machine language program it was given the name "TOP'N TAIL". Source listings of TOP and TAIL appear in Appendix III and the example listed in Appendix V uses them.

9.0 TESTING AND DEBUGGING WITH THE MONITOR

The MONITOR is the machine language monitor for your computer. It 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 Activating the MONITOR

The machine language monitor on the C64 Assembler is accessed by typing "MON" and pressing RETURN. The C128 and Plus 4 have built in monitors which function in a similar manner to the C64 monitor; the user is advised to read the appropriate Commodore documentation for full details.

9.2 Using the C64 MONITOR

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

9.3 MONITOR Program Commands

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 in the line of program execution to prevent loss of control.)

Example: G 040C

Execution begins at location 040C.

COMMAND: L (LOAD)

Purpose : Load a file from disk.

Syntax : L "filename",(device)

filename: Any legal filename

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

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

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 PGM 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", 08 :reads a file from disk drive.

COMMAND: M (MEMORY DISPLAY)

Purpose : To display memory as a hexadecimal dump within the specified address range. In addition, the ASCII equivalent of each byte is shown at the end of each line of display.

Syntax : M(address 1)(address 2)

(address 1): First address of the hex dump

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

Memory is displayed in the following format:

```
ADDRESS     CONTENT              ASCII

.: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.

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

The first eight bytes of memory are displayed.

COMMAND : R (REGISTER DISPLAY)

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

Syntax : R

Example : R
```
     PC   IRQ   SR  AC  XR  YR  SP
;  0850  96C8  02  03  04  FE  F4
```

COMMAND : S (SAVE)

Purpose: Save the contents of memory onto tape or disk.

Syntax : S "filename",(device),(address1),(address2)

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

(device): Any IEEE device or cassette may be used. The device number of the Commodore disk drives are normally factory set to 08. Two digits must always be entered, for example 08 for device 8 and 01 for the cassette drive.

(address 1): Starting address of memory to be saved, always 4 digits.

(address 2): Four digits specifying the 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 $0BFF onto disk.

COMMAND : H (HUNT)

Purpose : Searches through a specified memory range for the occurence of a string of up to 40 hex values.

Syntax : H (address1) (address2) (b1) (b2).....

(address1) : start address of search.
(address2) : end address of search.
(b1) (b2) : hexadecimal values of the string to be located.

Example : H 2000 3500 20 D2 FF

Hunts from the start address $2000 to the end address $3500 for the string of three bytes $20 $D2 $FF. The address of each location where the match is found is displayed as a four byte hexadecimal value.

Hunt may be terminated with the stop key.

Command : F (FILL)

Purpose : Fill a specified area of memory with a single byte value.

Syntax : F (address1) (address2) (byte)

(address1) : First address to be filled.
(address2) : Last address to be filled.
(byte) : Value to be written into memory.

Example : F 0400 0420 A0

Fills the address range $0400 to $0420 with the value $A0.

COMMAND : @ (READ/WRITE TO DISK)

Purpose : Allows the disk error channel to be read, and commands to be sent to the disk unit.

Syntax : @

Acheives the same function as the "@" in DOS Support. A channel is established to read the error channel on device 8 and the result displayed on the screen.

Syntax : @I0 . @N1:(disk name,id) @UJ etc

Transmits the string following the initial "@" to the disk command channel.

COMMAND : X (EXIT TO COMMAND LEVEL)

Purpose : Returns control to the machine to command level in BASIC or editor mode.

Syntax : X

COMMAND: D (DIS-ASSEMBLE FROM CORE) See note 1.

Purpose : Provides a screen dis-assembly from memory.

Syntax : D (address)

The screen dis-assembler will output ten lines of display and then wait.If the user presses the cursor down key more code will be shown, any other key results in the monitor prompt. Each line of display is in the following format:

$addr mne operand ;h1 h2 h3 a1 a2 a3

Where :

addr = the address of the opcode.
mne = mnemonic of the opcode.
operand = value of byte or bytes comprising the operand.
h1 h2 h3 = hex values of the opcode and associated operand.
a1 a2 a3 = ASCII equivalent of h1 h2 and h3.

Bytes which do not contain a valid 6500 opcode, which by virtue of their address would be expected to contain one, are displayed in .byt format :

$addr .byt $xx ;h1 a1

Where xx is the byte that does not dis-assemble as an opcode. Note that the dis-assembler uses the same mnemonic table as the assembler and thus will correctly display any special opcodes that have been patched by the operator.

Branch instructions are displayed specially :

$addr mne *+2+$xx ;$addr2 ;h1 h2 a1 a2

Where :
^ represents the address of the opcode.
+2 = the number of bytes taken up by the opcode and operand.
+$xx = the branch range AFTER the instruction has executed.
$addr2 = the target address of the branch instruction.

COMMAND : A (ASSEMBLE)

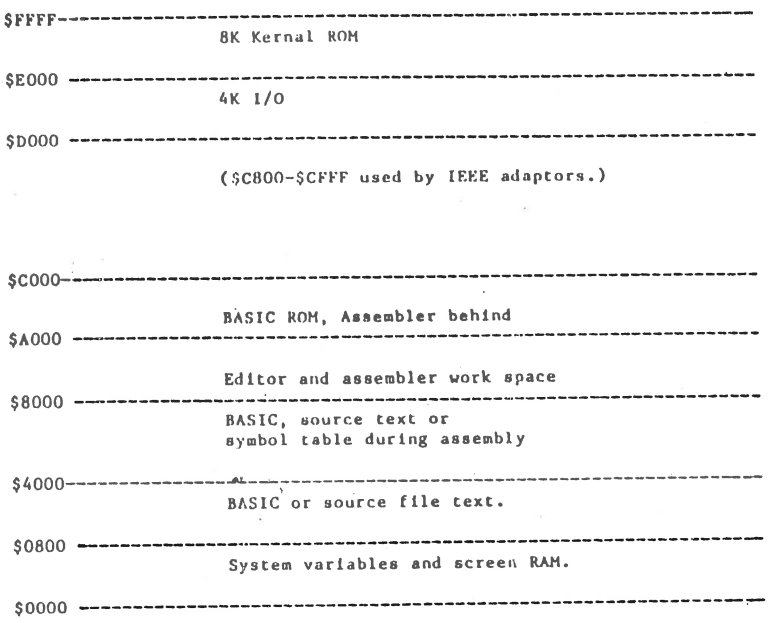
Purpose : Provides a simple line assembler which allows code to be assembled at a specified starting address.

Syntax : A (address)

The assembler prompts with the address specified in the original call. Enter source text using the standard 6500 mnemonics and operand format. Labels may not be used in a simple line assembler.

Branch references should be entered with dummy target address until the code is completed. Then patch the correct address. This procedure is simplified by using the dis-assembler which produces a screen display that may be immediatly re-assembled by entering the assembler with a dummy address, ammending the dis-assembled line of code and pressing return.

Appendix I MEMORY MAP FOR C-64 WITH JCL ASSEMBLER



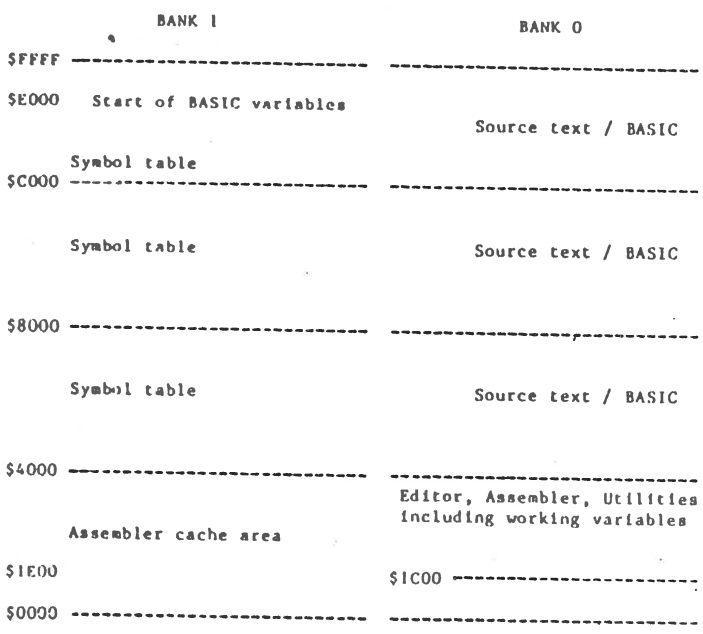
NOTES

1. The area $C000 to $CFFF is free for the testing of small programs without adjusting any system pointers.

2. Source/BASIC may occupy the space $0800 to $8000. 'SOURCE AT RISK' will be figged if the assembler is called and the current text exceeds $4000.

3. Several IEEE-488 bus adaptors use the area $C800 to $CFFF for software and storage so this space will become un-safe. The JCL Software adaptor works with all versions of the C-64 assembler when used with an appropriate mother board.

Appendix I MEMORY MAP FOR C128 Assembler

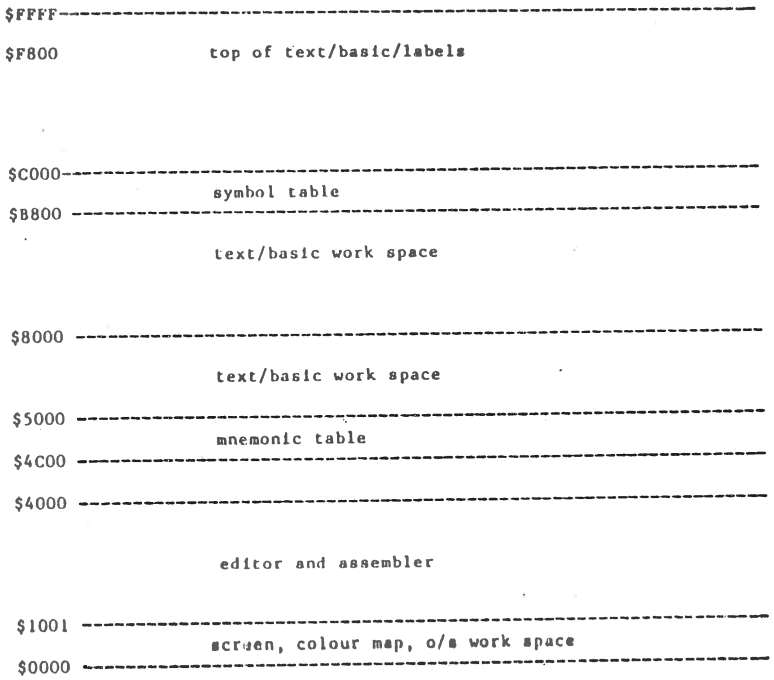


NOTES

1. The ASM command swaps the assembler from bank 1 to bank 0. All normal exit modes cause the Editor to be returned to bank 0.

2. The start of BASIC variables in bank 1 is set by the loader program. Users requiring more space for BASIC program development should make appropriate adjustments to thje loader.

Appendix I MEMORY MAP FOR PLUS 4 with Assembler



NOTES

1. The space $5000 to $F800 is available for development of source code or BASIC. 'SOURCE AT RISK' will be flagged if the source is larger than $B800 and the assembler is called.

Appendix II USING COMMAND FILES

In this context a command file is a BASIC program that mangages one or more of the steps used to convert source files into a program file residing on disk or running in memory. Command files automate repetitative tasks.

For example, take the development of a piece of code intended to deal with a communications task via the C64 user port. The source file is called "usercoms" and is stored on disk. It produces a loadable program using the "10 SYS(2063)" method of interfacing with BASIC. To re-assemble the software the following steps would be made manually:-

1. Erase old .MOD and .SYM files to ensure disk space is adequate.
2. Assemble the program using ASM "USERCOMS,M"
3. If there were no assembly errors then load the .MOD file into memory.
4. LOAD and RUN the completed program.

All versions of the assembler have an entry point which may be called with a SYS call from a BASIC program and an error flag in page zero which may be inspected with PEEK after the assembler has finished.

The following two points must be observed :

1. C64 and Plus 4 Command Files should not create BASIC variables unless MEMTOP is lowered to a point just above the command file, leaving sufficient space for the wanted variables.

2. The SYS call must give the source file name directly WITHOUT a drive number, drive 0 will be assumed.

Here is a typical C64 command file to automate the above steps.

```
10 REM SAVE"@0:CMD",8
20 OPEN 15,8,15,"S0:USERCOMS.*" : CLOSE15
30 SYS 32771, "USERCOMS,M"
40 IF PEEK (253) THEN STOP
50 LOAD "USERCOMS.MOD",8
```

When this program has been saved on disk, all that is necessary to re-assemble and run the machine language again is to RUN "CMD".

Examples of skeleton command files for each machine will be found on the Assembler system disk.

Appendix III TOP'N TAIL SOURCE LISTING FOR C-64

```
1000 ;PUT"@0:TOP"
1010 ;-------------------------------------
1020 ;(C)JCL SOFTWARE 'TOP AND TAIL'
1030 ;BASIC AND MACHINE LANGUAGE INTEGRATION
1040 ;
1050 ;SEE HS-64 USER MANUAL SECTION 8.3
1060 ;-------------------------------------
1070 .SKI 3
1080 TXTPTR = $7A             ;CHRGET POINTER
1090 TXTTAB = $2B             ;POINTER TO START OF BASIC
1100 ;
1110 * = $0801                ;NORMAL START OF BASIC
1120 ;
1130 ;SYNTHESIZE '10 SYS (2063)'
1140       .WORD REF          ;FORWARD POINTER
1150       .WORD 10           ;LINE #10
1160       .BYT $9E           ;'SYS' TOKEN
1170       .BYT '(2063)'
1180       .BYT 00            ;END OF BASIC LINE
1190 REF   .BYT 00,00         ;END OF TEXT DOUBLE NULL
1200 ;
1210 ;RESET BASIC TXTPTR ABOVE RESERVED SPACE
1220       LDA #<NULL
1230       STA TXTPTR
1240       LDA #>NULL
1250       STA TXTPTR+1
1260 ;
1270 ;RESET START OF BASIC
1280       LDA #<BASIC
1290       STA TXTTAB
1300       LDA #>BASIC
1310       STA TXTTAB+1
1320 ;
1330 ;JUMP OVER RESERVED SPACE
1335       JMP TAIL
1340 ;
1350 ;PUT JUMP TABLE FOR USER CODE HERE.
1360 ;CONCLUDE USER SOURCE WITH .LIB TAIL
1370 .END
```

```
1000 ;PUT "@0:TAIL"
1010 ;-------------------------------------
1020 ;(C)JCL SOFTWARE 'TOP AND TAIL'
1030 ;BASIC AND MACHINE CODE INTEGRATION
1040 ;SEE HS64 USER MANUAL SECTION 8.3
1050 ;-------------------------------------
1060 .SKI 3
1070 TAIL   RTS               ;RETURN TO BASIC
1080 NULL   .BYT 00           ;NULL AT START OF BASIC
1090 BASIC  .BYT 00,00        ;2 NULLS = ZERO PROGRAM
1100 ;
1110 .END
```

Note that the example in Appendix V uses TOP'N TAIL

Appendix IV 6500 SERIES MICROPROCESSOR INSTRUCTION SET

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 result zero
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 interupt or break
BVC Branch on overflow clear
BVS Branch on overflow set
CLC Clear carry flag
CLD Clear decimal mode
CLI Clear interupt 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 interupt
RTS Return from subroutine
SBC Subtract memory and carry from accumulator
SEC Set carry flag
SED Set decimal mode
SEI Set interupt 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 to index X
TXA Transfer index X to accumulator
TXS Transfer index X to stack register
TYA Transfer index Y to accumulator

Appendix V A SAMPLE LISTING FROM THE JCL SOFTWARE ASSEMBLER

dump screen.....page # 2

```
Line#  Addr Code         Source

00003  0000             ;call "top" with .lib and no list
00005  0000             .lib top
00046  0822             .opt list


00048  0822             ;---------------------------------
00049  0822             ;routine to copy screen to current
00050  0822             ;logical file number 4. basic uses
00051  0822             ;a sys call to perform dump :-
00052  0822             ; 100 open4,4 : sys2082 : close4
00053  0822             ;---------------------------------


00055  0822             frekzp   = $fb           ;zero page pair
00056  0822             chkout   = $ffc9         ;open o/p channel
00057  0822             clrchn   = $ffcc         ;close channel
00058  0822             chrout   = $ffd2         ;ieee output
00059  0822             screen   = $0400         ;screen ram origin

00060  0822             ;
00061  0822             ;- entry point for sys call -
00062  0822             ;
00063  0822 a2 00        entry  ldx #<screen      ;make pointer to screen
00064  0824 a9 04              lda #>screen
00065  0826 86 fb              stx frekzp
00066  0828 85 fc              sta frekzp+1
00067  082a             ;
00068  082a a2 04              ldx #4           ;open output channel
00069  082c 20 c9 ff           jsr chkout
00070  082f             ;
00071  082f a9 19              lda #25          ;counter for screen lines
00072  0831 8d 7a 08           sta temp2
00073  0834             ;
00074  0834 a9 11        loopa  lda #17          ;cbm printer line preamble
00075  0836 20 d2 ff           jsr chrout
00076  0839             ;
00077  0839 a0 00              ldy #0           ;line index
00078  083b b1 fb        loopb  lda (frekzp),y   ;get chr from the screen
00079  083d 20 63 08           jsr trans        ;make into ascii
00080  0840 20 d2 ff           jsr chrout       ;output to lf - 4
00081  0843 c8                 iny              ;bump line scan
00082  0844 c0 28              cpy #40          ;test for limit
00083  0846 d0 f3              bne loopb        ;not yet...
00084  0848 a9 0d              lda #$d          ;send a c/r
00085  084a 20 d2 ff           jsr chrout
00086  084d                    ;
00087  084d                    ;
00088  084d ce 7a 08           dec temp2        ;dec line count
00089  085d f0 0d              beq exit         ;exit if done
00090  0852                    ;
```

```
dump screen.....page # 3

Line#  Addr Code        Source


00091                  ;bump screen pointer to next line
00092  0852 a5 fb              lda frekzp
00093  0854 18                 clc
00094  0855 69 28              adc #40
00095  0857 85 fb              sta frekzp
00096  0859 90 d9              bcc loopa
00097  085b e6 fc              inc frekzp+1
00098  085d d0 d5              bne loopa
00099  085f             ;
00100  085f 20 cc ff    exit   jsr clrchn          ;close channel
00101  0862 60                 rts                ;return to basic
00102  0863             ;
00103  0863             ;subrtn to convert screen code to cbm ascii
00104  0863 29 7f       trans  and #$7f
00105  0865 8d 79 08           sta templ
00106  0868 29 3f              and #$3f
00107  086a 0e 79 08           asl templ
00108  086d 2c 79 08           bit templ
00109  0870 10 02              bpl tr1
00110  0872 09 80              ora #$80
00111  0874 70 02       tr1    bvs tr2
00112  0876 09 40              ora #$40
00113  0878 60                 rts
00114  0879             ;
00115  0879             ;temporary storage area
00116  0879             templ * = *+1
00117  087a             temp2 * = *+1
00118  087b             ;
00119  087b             ;call 'tail' with .lib and no list
00121  087b             .lib tail
00135  087f             .opt list

end of assembly, error count = 00000

basic   087d    chkout  ffc9    chrout  ffd2    clrchn  ffcc
entry   0822    exit    085f    frekzp  00fb    loopa   0834
loopb   083b    null    087c    ref     080d    screen  0400
tail    087b    templ   0879    temp2   087a    tr1     0874
tr2     0878    trans   0863    txtptr  007a    txttab  002b
```


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.

**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 correct, restructure the program.

**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 label, 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 or module file name already exists on the diskette. This error can be corrected by scratching the old file or changing the diskette.

Note that the Assembler opens output files BEFORE starting PASS 1, so little time is wasted if the file already exists. This is preferable to the more usual practice of opening output files at the start of PASS 2, which can be after some time has elapsed.

**FILE NOT FOUND

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

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

A label or expression which uses a yet undefined value is considered to be referenced forward to the to-be-defined value and results in the use of a

dummy value on pass 1.

**ILLEGAL ADDRESS MODE

After finding an opcode that does not have an implied operand, the assembler parses 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).

The assembler recognizes an indirect address by the 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 message will be printed.

**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 as 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 line). 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.

**INDEX 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 interprets the next field as an indirect address 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).

**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 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 line with a long first word; and doesn't begin with a semi-colon. 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 lone label or an opcode that needs no operand, they can be followed directly by a semicolon to denote the rest of the line is a comment.

**PC DECREMENT

This error is considered to be a fatal error and assembly terminates if a module file is being produced. The error is generated when the program counter is decremented due to incrementing through $FFFF or when a new origin statement redefines the program counter to an address lower thn that expected for the next byte.

Module files are directly loadable program files and carry a load address specified by the first byte of assembled code. Forward discontinuities are detected and the intervening space filled with $FF bytes. Negative discontinuities are not allowed.

Hex object code files will accept a negative transition because each time an address discontinuity occurs the current output line is terminated, and a new line started with the new program counter value. This is tolerable, but bad practice. (Note that the JCL Software EPROM programmer will also tolerate this practice.)

**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.

**SOURCE AT RISK *NOT C128*

The symbol table used by the assembler and source code being editted share the same address space. In the C64 about 14K of source code may be retained in memory without risk of overwriting by the symbol table. (Plus4 - 26K.) When the assembler is used it first checks to see if the end of the text held in memory is above the start of the symbol table, and if it is then this error message is displayed and the assembler returns to command level. The corrective procedure is to save the text using PUT, and then use NEW to clear memory.

**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.

**MISSING/ILLEGAL SYMBOL

This error is generated by the second pass. If a symbol is defined (shows up on the left of an equate or as a label in a statement), pass one will 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.

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 operand.

This error message will also occur if the assembler is looking for a needed field and runs off the end of the line before the field is found.

VERY IMPORTANT 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 reserve two bytes for the value of the expression.
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.

IT IS THUS ESSENTIAL THAT ALL ZERO PAGE REFERENCES ARE DECLARED BEFORE THEY ARE REFERRED TO IN AN OPERAND.

The good programming practice of declaring all variables and memory assignments at the start of the source file will avoid any problems from this source.

Appendix VII EDITOR COMMAND SUMMARY

NOTE: Detail varies from machine to machine. Use the command 'WORDS' to check which options are in the version you are currently using.

Command		Description
ASCII		Select ASCII printer mode for ASM and TYPE
ASM		Call the Assembler system.
AUTO	*	Terminates AUTO nl
BASIC		Exit EDIT mode and return to BASIC
BOOM	*	Call cold start vector
BORDER nl	*	Select border colour
CBM		Select standard CBM printer mode
CHANGE/s1/s2	*	Change string throughout entire text
CHANGE/s1/s2/,n1-n2	*	Change string in line range
CLRBRK		Restore BRK set by SETBRK
DEL n1-n2	*	Delete a range of lines
DH nl		Display hex equivalent of nl
DO	*	Execute first line of text/program
EDIT		Select EDIT mode of operation
FIND/s1/	*	Find string throughout entire text
FIND/s1/,n1-n2	*	Find string in line range
GET "FILE"		Load source file
GET "FILE",nl		Append file at line nl
HD xxxx		Display decimal equivalent of four digit hex string
JOIN "PROGRAM"	*	Concatenate program to BASIC
MON	*	Call the Monitor
PUT "dr:FILE"		Save text on disk file
PUT "dr:FILE",n1-n2		Save part of text line range n1-n2
RBAS	*	Reset start of BASIC pointer
REN n1,n2,n3	*	Renumber new start, step, old start

SCREEN nl	*	Select screen background colour
SCROLL	*	Turn on scroller and screen re-write
SETBRK $xxxx		Set a BRK at 4 digit hex address xxxx
SIZE "PROGRAM"	*	Display start and end address of program
TYPE	*	Display text on screen in pages format
TYPE "FILE"	*	Display file on screen
TYPE options	*	u4=printer lxx=page length hxx=head+foot lines mxx=margin width
WORDS	*	Provide a listing of all EDITOR commands

SCROLL OPTIONS

F1 List from line displayed at top of screen. Plus4 - F1
F3 List from line indicated by the cursor. Plus4 - F2
F5 List from bottom line of the screen. Plus4 - F3
F7 E erase to end of line.
S erase to start of line.
T erase to top of screen.
F erase to foot of screen.
I insert a line after current cursor line.
X exit, no action.

Commands marked with '*' may be used in BASIC or EDIT modes of operation.

See sections 6.3 and 6.4 for a full description of the operation of these commands.

Appendix VIII C64 MONITOR COMMAND SUMMARY

Command	Format	Description
LOAD	L "dr:filename",dv	Load file from device dv drive dr into memory
MEMORY	M add1 add2	Display memory between limits add1 and add2
FILL	F add1 add2 byt	Fills memory between add1 and add2 with the value specified by byt.
HUNT	H add1 add2 byt1 byt2...	Searches memory between add1 and add2 for the string byt1, byt2 etc.
REGISTERS	R	Display the CPU registers with option to ammend.
SAVE	S "dr:filename",dv,add1,add2	Save memory on device dv drive dr starting at add1 and ending at add2-1.
EXIT	X	Return to normal command level.(BASIC or EDIT mode)
DISK	@	Read and report disk error channel.
GO	G add1	Start execution at add1
GO	G	Start execution, status as set by R (Registers).
DISASSEMBLE	D addr	Start dis-assembler display at specified address. Press cursor down for more.
ASSEMBLE	A addr	Enter line assembler.

See section 9.3 for a complete description of these commands.

Appendix IX C64 DOS SUPPORT COMMAND SUMMARY

Note that the Plus 4 and C128 have equivalent functions built in as standard so the DOS Support facilities are only included in the C64 version.

In all cases the @ symbol may be raplaced by the > symbol

Command	Description
@	Read and display current disk status.
@C(dr):newfile=(dr):oldfile	Copy files.
@I(dr)	Initialise drive dr.
@N(dr):diskname(,id)	Format a disk.
@R(dr):nename=oldname	Rename a file.
@S(dr):filename	Scratch a file.
@UJ	Reset disk unit.
@$(dr):partname(*)	Display disk directory showing files commencing with partname
/filename	Load a BASIC program.
^filename	Load a BASIC program and RUN.
Xfilename	Load a program file at its own address.
Xfilename,addr	Load a program file at the decimal address specified.

See section 5.3 for a complete description of each of these commands.

Appendix X USING THE ASSEMBLER WITH OTHER 6500 SERIES PROCESSORS

The 6500 family of processors used in Commodore and other computers and single board controllers, are manufactured world wide by a number of companies. Some members of the family have expanded instruction sets that include, for example the ability to push and pull the X and Y registers to and from the stack in addition to the accumulator.

The assembler allows additional opcodes to be added to the standard instruction set and so makes the Commodore computer an ideal low cost program development sytem for these later processors.

During assembly each opcode mnemonic is compared with a table of mnemonics and address mode codes to determine the correct opcode to assemble. This table is located in RAM. The table consists of a four byte cell for each opcode and includes a dummy cell for each instruction that is missing from the standard 6502 set, making a total of 1024 bytes. The cells are stored in opcode order and thus un-used cells may be over written by using the monitor, or more conveniently a BASIC program.

Each four byte cell is made up of the opcode mnemonic (3 letters, no more, no less) followed by an address mode code. The address mode codes are as follows :-

Address Mode	Code	Example
Indexed indirect	48	LDA (label,X)
Zero page	49	LDA $42
Immediate	50	LDA #42
Absolute	51	LDA $4242
Indirect indexed	52	LDA (label),Y
Zeropage,X	53	LDA $42,X
Absolute,Y	54	LDA $4242,Y
Absolute,X	55	LDA $4242,X
Absolute indirect	56	JMP ($0300)
Zero page,Y	57	LDA $42,Y
Accumulator	58	ROL A
Implied	59	INX
Relative	60	BNE label

A skeleton example for each machine will be found on the system disk.