MERLIN 64™

The Macro Assembler
For The Commodore 64

By Glen Bredon

INSTRUCTION MANUAL

ISBN 0-927796-05-8

PRODUCED BY:

Roger Wagner™
PUBLISHING, INC.

10761 Woodside Avenue • Suite E • Santee, California 92071
Customer Service & Technical Support: 619/562-3670

OUR GUARANTEE

This product carries the unconditional guarantee of satisfaction or your money back. Any product may be returned to place of purchase for complete refund or replacement within thirty (30) days of purchase if accompanied by the sales receipt or other proof of purchase.

ROGER WAGNER PUBLISHING, INC.

CUSTOMER LICENSE AGREEMENT

IMPORTANT: The Roger Wagner Publishing, Inc. software product that you have just received from Roger Wagner Publishing, Inc., or one of its authorized dealers, is provided to you subject to the Terms and Conditions of the Customer License Agreement. Should you decide that you cannot accept these Terms and Conditions, then you must return your product with all documentation and this License marked "REFUSED" within the 30 day examination period following receipt of the product.

1. License. Roger Wagner Publishing, Inc. hereby grants you upon receipt of this product, a nonexclusive license to use the enclosed Roger Wagner Publishing, Inc. product subject to the terms and restrictions set forth in this License Agreement.

2. Copyright. This software product, and its documentation, is copyrighted by Roger Wagner Publishing, Inc. You may not copy or otherwise reproduce the product or any part of it except as expressly permitted in this License.

3. Restrictions on Use and Transfer. The original and any backup copies of this product are intended for your personal use in connection with a single computer. You may not distribute copies of, or any part of, this product without the express written permission of Roger Wagner Publishing, Inc.

LIMITATION ON WARRANTIES AND LIABILITY

ROGER WAGNER PUBLISHING, INC. AND THE PROGRAM AUTHOR SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO PURCHASER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY THIS SOFTWARE, INCLUDING, BUT NOT LIMITED TO ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THIS SOFTWARE. SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES OR LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

ABOUT THE AUTHOR

Glen Bredon is a professor at Rutgers University in New Jersey where he has taught mathematics for over fifteen years. He purchased his first computer in 1979 and began exploring its internal operations because "I wanted to know more than my students." The result of this study was the best selling Merlin Macro Assembler and other programming aids. A native Californian and concerned environmentalist, Glen spends his summers away from mathematics and computing, prefering the solitude of the Sierra Nevada mountains where he has helped establish wilderness reserves.

PRODUCT REFERENCE:MERLIN5C0784LC

TABLE OF CONTENTS

INTRODUCTION

Assembly Language Whys and Wherefores

Some of you may ask "What is Assembly Language?" or "Why do I need to use Assembly Language; BASIC suits me fine." While we do not have the space here to do a treatise on the subject, we will attempt to briefly answer the above questions.

Computer languages are often referred to as "high level" or "low level" languages. BASIC, COBOL, FORTRAN and PASCAL are all high level languages. A high level language is one that usally uses English-like words (commands) and may go through several stages of interpretation or compilation before finally being placed in memory. The time this processing takes is the reason BASIC and other high level languages run far slower than an equivalent Assembly Language program. In addition, it normally consumes a great deal more available memory.

From the ground up, your computer understands only two things, on and off. All of its calculations are handled as addition or subtraction but at tremendously high speeds. The only number system it comprehends is Base 2 (the Binary System) where a 1 for example is represented by ØØØØØØØ1 and a 2 is represented by ØØØØØØ1Ø.

The 651Ø microprocessor (the 651Ø is equivalent to the 65Ø2) has five 8-bit registers and one 16-bit register. All data is ultimately handled through these registers by a machine language program. But even this lowest of low-level code requires a program to function correctly. This "program" is hard wired within the 651Ø itself. The microprocessor program functions in three cycles: it fetches an instruction from memory in the computer, decodes it and executes it.

These instructions exist in memory as one, two or three byte groups. A byte contains 8 binary bits of data and is usually notated in hexadecimal (base 16) form. Some early microcomputers allowed data entry only through 8 front panel switches, each of which when set on or off would combine to produce one binary byte. This required an additional program in the computer to monitor the switches and store the byte in memory so that the 651Ø could interpret it.

At the next level up, the user may enter his/her data in the form of a three character mnemonic (the "m" is silent), a type of code whose characters form an association with the microprocessor operation. For example: LDA is a mnemonic which represents "LoaD the Accumulator". The Commodore monitor cartridge add-on has a mini-assembler that permits simple Assembly Language programming.

But even this is not sufficient to create a long and comprehensive program. In addition to the use of a three character mnemonic, a full fledged assembler allows the programmer to use labels, which represent an as yet undefined area of memory where a particular part of the program will be stored. In addition, an assembler will have a provision for line numbers, similar to those in a BASIC program, which in turn permits the programmer to insert lines into the program and perform other editing operations. This is what MERLIN is all about.

Finally, a high level language such as BASIC is itself an assembly program which takes a command such as PRINT and reduces it by tokenizing to a single byte before storing it in memory.

Before using this or any other assembler, the user is expected to be somewhat familiar with the 651Ø architecture, modes of addressing, etc. This manual is not intended to teach Assembly Language programming. Many good books on 651Ø assembly programming (which is nearly identical to 65Ø2 assembly programming) are available at your local bookstore.

SYSTEM REQUIREMENTS

* 64k Commodore 64
* Disk Drive

OPTIONAL:

* DATA2Ø Videopack8Ø
* VIC-1Ø11A RS232 cartridge

BEGINNERS GUIDE TO USING MERLIN

By T. Petersen

Notes and demonstrations for the beginning MERLIN programmer.

Introduction

The purpose of this section is not to provide instruction in assembly language programming. It is to introduce MERLIN to programmers new to assembly language programming in general, and MERLIN in particular.

Many of the MERLIN commands and functions are very similar in operation. This section does not attempt to present demonstrations of each and every command option. The objective is to clarify and present examples of the more common operations, sufficient to provide a basis for further independent study on the part of the programmer.

A note of clarification:

Throughout the MERLIN manual, various uses are made of the terms "mode" and "module".

In this section, "module" refers to a distinct computer program component of the MERLIN system. There are five MODULES:

1. The EXECUTIVE
2. The EDITOR
3. The ASSEMBLER
4. The SYMBOL TABLE GENERATOR
5. The MONITOR

Each module is grouped under one of the three CONTROL MODES:

1) The EXECUTIVE, abbreviated EXEC and indicated by the '%' prompt.

2) The EDITOR, indicated by the ':' prompt.

3) The MONITOR, indicated by the '$' prompt.

EXECUTIVE CONTROL MODE
Executive Module

EDITOR CONTROL MODE
Editor Module
Assembler Module
Symbol Table Generator Module
Monitor Module

The term "mode" may be used to indicate either the current control mode (as indicated by the prompt) or alternatively, while in control mode and subsequent to the issuance of an entry command, the system is said to be in '[entry command] mode'. For example, while typing in a program after issuing the ADD command, the system is said to be 'in ADD mode'.

Terminating [entry command] mode returns the system to control mode.

Input

Programmers familiar with some assembly and higher-level languages will recall the necessity of formatting the input, i.e. labels, opcodes, operands and comments must be typed in specific fields or they will not be recognized by the assembler program.

In MERLIN, the TABS operator provides a semi-automatic formatting feature.

When entering programs, remember that during assembly each space in the source code causes a tab to the next tab field. As a demonstration, let's enter the following short routine.

Steps from the very beginning:

1. LOAD "MERLIN",8 and RUN

2. When the '%' prompt appears at the bottom of the EXEC mode menu, type 'E'. This instantly places the system in EDITOR control mode.

3. Since we are entering an entirely new program, type 'A' at the ':' prompt and press RETURN (A = ADD). A '1' appears one line down and the cursor is automatically tabbed one space to the right of the line number. The '1' and all subsequent line numbers which appear after the RETURN key is pressed serve roughly the same purpose as line numbers in BASIC except that in assembly source code, line numbers are not referenced for jumps to subroutines or in GOTO-like statements.

4. On line 1, enter an '*' (asterisk). An asterisk as the first character in any line is similar to a REM statement in BASIC - it tells the assembler that this is a remark line and anything after the asterisk is to be ignored. To confirm this, type the title 'DEMO PROGRAM 1' after the asterisk and hit the RETURN key.

5. After return, the cursor once again drops down one line, a '2' appears and the cursor skips a space. Now type a space, type 'ORG', space, '$8000', RETURN.

6. On line 3, type 'CHROUT', space, 'EQU', space, '$FFD2', and RETURN.

7. On line 4, type 'START', space, 'LDA', space, '#$93', and RETURN.

8. Line 5 - type a space, 'JSR', space, 'CHROUT', space, ';' (semicolon), 'CLEAR THE SCREEN', RETURN. Semicolons are a convention often used within command lines to mark the

start of comments.

9. Line 6 - 'DONE', space, 'RTS', RETURN.

10. The program has been completely entered, but the system is still in ADD mode. To exit ADD, just press RETURN. The ':' prompt reappears at the left of the screen, indicating that the system has returned to control mode.

11. The screen should now appear like this:

```
 1 *DEMO PROGRAM 1
 2        ORG   $8000
 3 CHROUT EQU   $FFD2
 4 START  LDA   #$93
 5        JSR   CHROUT          ;CLEAR THE SCREEN
 6 DONE   RTS
```

Note that each string of characters has been moved to a specific field. There are four such fields, not including the line numbers on the left.

Field number...

One is reserved for labels. CHROUT, START and DONE are examples of labels.

Two is reserved for opcodes, such as the MERLIN pseudo-ops ORG and EQU, and the 6510 opcodes LDA, JSR and RTS.

Three is for operands, such as $8000, $FFD2 and, in this case, CHROUT.

Four will contain any comments.

It should be apparent from this exercise that it is not necessary to input extra spaces in the source file for formatting purposes.

In summary, after the line numbers:

1) Do not space before a label. Press space once after label (or if there is no label, once after the line number) for the opcode.
2) Space once after the opcode for the operand. Space once after the operand for the comment. If there is no operand, type a space and a semicolon.

System and Entry Commands

MERLIN has a powerful and complex built-in editor. Complex in the range of operations possible but, after a little practice, remarkably easy to use.

The following paragraphs contain only minor clarifications and brief demonstrations on the use of both sets of commands. All System and Entry commands are used in EDITOR Control Mode immediately after the ':' prompt.

The STOP key or RETURN as the first character of a line exits the current [entry command] mode and returns the system to control mode when ADDing or INSERTing lines. STOP exits edit mode and returns the system to control mode after Editing lines.

The other System and Entry Commands are terminated either automatically or by pressing RETURN.

Inserting and deleting lines in the source code are both simple operations. The following example will INSERT three new lines between the existing lines 5 and 6.

1. After the ':' prompt, type 'I' (INSERT), the number '6', and press RETURN. All inserted lines will precede (numerically) the line number specified in the command.

2. Input an asterisk, and press RETURN. Note that INSERT mode has not been exited.

3. Repeat step 2.

4. Type one space, 'TYA', and press RETURN.

   On the screen is the following:

```
:I6
   6 *
   7 *
   8     TYA
   9
```

5. Hit RETURN (or STOP) and the system reverts to CONTROL mode (':' prompt).

6. LIST the source code.

```
:L
     1  *DEMO PROGRAM 1
     2         ORG   $8000
     3  CHROUT EQU   $FFD2
     4  START  LDA   #$93
     5         JSR   CHROUT     ;CLEAR THE SCREEN
     6 *
     7 *
     8         TYA
     9  DONE   RTS
```

The three new lines (6, 7, and 8) have been inserted, and the subsequent original source lines (now line 9) have been renumbered.

Using DELETE is equally easy.

1. In control mode, input 'D8', and RETURN. Nothing new appears on the screen.

2. LIST the source code. The source listing is one line shorter. You've just deleted the 'TYA' line, and the subsequent lines have been renumbered.

It is possible to delete a range of lines in one step.

1. In control mode, input 'D6,7' and RETURN.

2. LIST the source.

Lines 6 and 7 from the previous example, which contained the inserted asterisk comments, have been deleted, and the subsequent lines renumbered. The listing appears the same as in the subsection on INPUT, Step 11.

This automatic renumbering feature makes it IMPERATIVE that when deleting lines you remember to begin with highest line number and work back to the lowest.

The Add, Insert, or Edit commands have several sub-commands comprised of CTRL-characters. To demonstrate, using our previous routine:

1. After the ':' prompt, enter 'E' (the EDIT command) and a line number (use '6' for this demonstration), and hit RETURN. One line down the specified line appears in its formatted state:

```
6 DONE    RTS
```

   and the cursor is over the 'D' in 'DONE'.

2. Type CTRL-D. The character under the cursor disappears. Type CTRL-D again, again, and a fourth time, 'DONE' has been deleted, and the cursor is positioned to the left of the opcode.

3. Hit RETURN and LIST the program. In line 6 of the source code, only the line number and opcode (RTS) remain.

4. Repeat step 1 (above).

5. This time, type CTRL-I (or the INSERT key). Don't move the cursor with the space bar or arrow keys. Type the word 'DONE', and RETURN.

6. LIST the program. Line 6 has been restored.

If you are editing a single line, hitting RETURN alone restores you to the control mode prompt. In step 1 (above), if you had specified a range of lines (example: 'E3,6') while issuing the EDIT command, RETURN would have called up the next sequential line number within the specified range. As the lines appear, you have the options of editing using the various sub-commands, pressing RETURN which will call up the next line, or exiting the EDIT mode using STOP. NOTE: hitting RETURN will enter the entire line in memory, exactly as it appears on the screen.

The other sub-commands, (CTRL-characters) used under the EDIT command function similarly. Read the definitions in Section 3 and practice a few operations.

Assembly

The next step in using MERLIN is to assemble the source code into object code.

After the ':' prompt, type the edit module system command ASM and hit return. On your screen is the following;

```
UPDATE SOURCE (Y/N)?
```

Type N, and you will see:

```
ASSEMBLING
                      1    *DEMO PROGRAM  1
                      2                ORG  $8000
                      3    CHROUT      EQU  $FFD2
8000     A9 93        4    START       LDA  #$93
8002     20 D2 FF     5                JSR  CHROUT;CLEAR THE SCREEN
8005     60           6    DONE        RTS

--END ASSEMBLY, 6 BYTES, ERRORS: 0
```

```
SYMBOL TABLE - ALPHABETICAL ORDER

     CHROUT    =$FFD2          ?   END     =$8ØØ5

 ?   START     =$8ØØØ

SYMBOL TABLE - NUMERICAL ORDER

 ?   START     =$8ØØØ          ?   END     =$8ØØ5

     CHROUT    =$FFD2
```

If instead of completing the above listing, the system beeps and displays an error message, note the line number referenced in the message, and press RETURN until the "END ASSEMBLy..." message appears. Then refer back to the subsection on INPUT and compare the listing with step 11. Look especially for elements in incorrect fields. Using the editing functions you've learned, change any lines in your listing which do not look like those in the listing in step 11 to what they should, then re-assemble.

If all went well, to the right of the column of numbers down the middle of the screen is the now familiar, formatted source code.

To the left of the numbers, beginning on line 4, is a series of numeric and alphabetic characters. This is the object code - the opcodes and operands assembled to their machine language hexadecimal equivalents.

Left to right, the first group of characters is the routine's starting address in memory (see the definition of ORG in the section entitled "Pseudo Opcodes - Directives"). After the colon are the numbers 'A9' and '93'. This is the one-byte hexadecimal code for the opcode LDA and the data byte $93.

NOTE: the label 'START' is not assembled into object code; neither are comments, remarks, or pseudo-ops such as ORG. Such elements are only for the convenience and utility of the programmer and the use of the assembler program.

The next byte is a 2Ø, which is the value for 'JSR'. The next two bytes (each pair of characters is one byte) on line 5 bear a curious resemblance to the last group of characters on line 3; have a look. In line 3 of the source code we told the assembler that the label 'CHROUT' EQUated with address $FFD2. In line 5, when the assembler encountered 'CHROUT' as the operand, it substituted the specified address. The sequence of the high and low-order bytes was reversed, turning $FFD2 into D2 FF, a 651Ø microprocessor convention.

The rest of the information presented should explain itself. The total errors encountered in the source code was zero, and six bytes of object code (count the bytes following the addresses) was generated.

Saving and Running Programs

The final step in using MERLIN is running the program. Before that, it is always a good idea to save the source code. Use the SAVE SOURCE command. Follow that with an OBJECT CODE SAVE. Note that OBJECT CODE SAVE must be preceded by a successful assembly.

1. Return to control mode if necessary, and type 'Q' RETURN. The system has quit EDITOR mode and reverted to EXECUTIVE (EXEC) mode. If the MERLIN system disk is still in the drive, remove it and insert an initialized work disk.

   After the '%' prompt, type 'S' (the EXEC mode SAVE SOURCE command). The system is now waiting for a filename. Type 'DEMO1', and RETURN. After the program has been saved, the prompt returns.

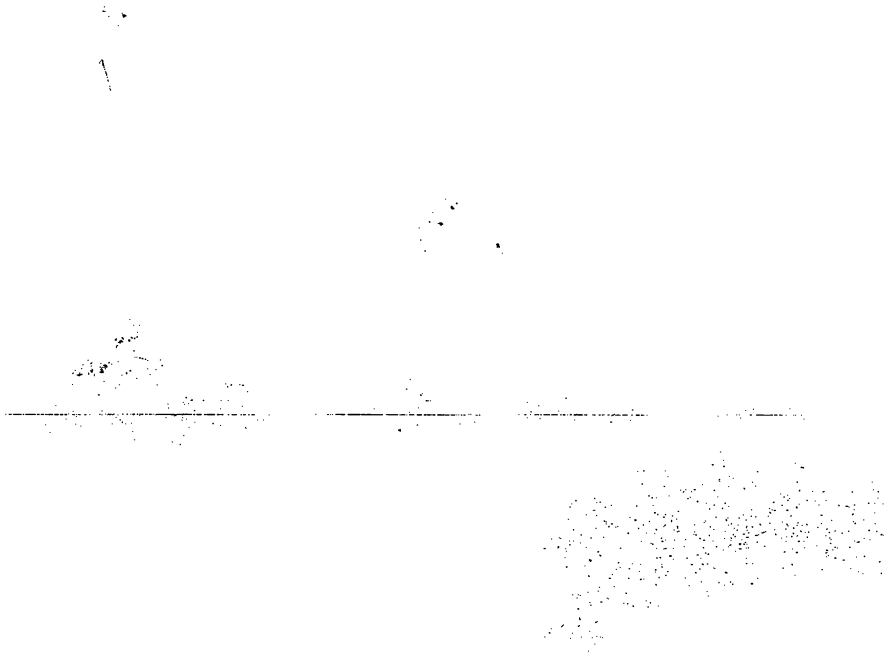
2. Type 'C' (CATALOG) and look at the disk catalog. The source code has been saved as a binary file titled "DEMO1.S". The suffix ".S" is a file-labelling convention which indicates the subject file is source code. This suffix is automatically appended to the name by the SAVE SOURCE command.

3. Hit RETURN to return to EXEC mode and input 'O', for OBJECT CODE SAVE. The object file should be saved under the same name as was earlier specified for the source file, so press "Y" to accept 'DEMO1' as the object name. There is no danger of overwriting the source file because the suffix '.O' is appended to object code file names.

While writing either file to disk, MERLIN also displays the address parameter, and the length parameter, which is the first address following the code. It's a good practice to take note of these.

Return to EDITOR mode, type 'MON', and RETURN and the monitor prompt '$' appears. Enter '8ØØØG', and RETURN. The screen should clear. The demonstration program was responsible for it!

Now you can return to the EXEC by typing 'Q' and hitting RETURN, or you can return to the editor with 'R' and RETURN.



EXECUTIVE MODE

The EXECUTIVE mode is the program level provided for file maintenance operations such as loading or saving code or cataloging the disk. The following sections summarize each command available in this mode.

C:CATALOG

Use the space bar to make the Catalog pause, and any other key to restart. Use the STOP key to abort the Catalog.

L:LOAD

This command is used to load a source file from disk. You will be asked for the name of the file. You should not append ".S" since MERLIN does this automatically. If you have hit "L" by mistake, just press the space bar then RETURN and the command will be cancelled without affecting any file in memory. This also applies to the other EXECUTIVE mode commands.

After a .LOAD SOURCE (or APPEND SOURCE) command, you are automatically placed in the editor mode, just as if you had hit "E". Subsequent LOAD SOURCE or SAVE SOURCE commands will display the last used filename. The cursor is placed on the first character and all the standard Commodore editing features are in effect. Pressing RETURN will accept the filename as displayed. The source will be automatically loaded at the correct address.

S:SAVE

Use this to save a binary source file to disk. As in the load command, you do not specify the suffix ".S" and you can hit RETURN to cancel the command. As with the LOAD SOURCE command above, the filename will then be displayed and you may type RETURN to SAVE the same filename, or substitute a new filename.

NOTE: when a SAVE is done in the EXECUTIVE mode (or via the SAV opcode, or Write, etc.), an error will result if the file already exists. This can be avoided by starting the filename with "@:" or "@0:". Thus, the replacement file is saved before the old file is scratched, so there must be room for the file on the disk.

This syntax may also be used for a LOAD but has no effect other than making the "@:" part of the default filename.

CAUTION: occasional serious disk problems may result from using the "@:" syntax and this is presumably due to a bug in the Commodore operating system. Thus, unless you have adequate back ups for your files, it is suggested that this syntax be avoided and, instead, that you scratch files before re-saving them.

A:APPEND

This loads in a specified source file and places it at the end of the file currently in memory. It operates in the same way as the LOAD SOURCE command, but does not affect the default file name. It does not save the appended file; you are free to do that if you wish.

D:DRIVE

This command will select the default disk device. The value here must be equal to or greater than eight.

E:ENTER ED/ASM

This command places you in the EDITOR/ASSEMBLER mode. It automatically sets the default tabs for the editor to those appropriate for source files. If you wish to use the editor to write an ordinary text file, you can type TABS and press RETURN to zero all tabs.

O:SAVE OBJECT CODE

This command is valid only after the successful assembly of a source file. In this case you will see the address and length of the object code on the menu. As with the source address, this is given for information only.

NOTE: that the object address shown is that of the program's ORG (or $8000 by default) and not that of the actual current location of the assembled code (which is $8000 or whatever HIMEM you have used). When using this command, you are asked for a name for the object file.

This command automatically appends an ".O" to the filename. Thus you can safely use the same name as that of the source file (without the ".S" of course). When this object code is saved to the disk its address will be the correct one, the one shown on the menu. When later, you LOAD it using the LOAD "FILE",8,1 syntax, it will go to that address, which can be anything ($800, $C000 etc.).

Q:QUIT

This exits to BASIC. You may re-enter MERLIN by issuing the "SYS 52000" command. This re-entry will be a warm start, which means it will not destroy the source file currently in memory.

R:READ TEXT FILE

This reads text files into MERLIN. They are always appended to the current buffer. To clear the buffer and start fresh, type "NEW" in the editor. If no file is in memory, the name given will become the default filename. Appended reads will not do this.

When the read is complete, you are placed in the editor. If the file contains lines longer than 255 characters, these will be divided into two or more lines by the READ routine. The file will be read only until it reaches HIMEM, will produce a memory error if it goes beyond, and only the data read to that point will remain.

W:WRITE TEXT FILE

This writes a MERLIN file into a text file instead of a binary file. The WRITE command does not delete or scratch first. See CAUTION above regarding "@:" syntax.

G:RUN PROGRAM

This command will LOAD and EXECUTE the named program. The program must be in machine language (not a BASIC program), and must have the ".O" suffix as part of the current program name. This provision is intended for using utilities such as "STRIP.O", "XREF.O" or for running "SOURCEROR.O".

X:DISK COMMAND

This sends the command to the "Error Channel". Examples of intended use are:

X then V will do a disk verify

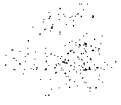
X then S:FILE.O will scratch "FILE.O"

X then R:NEWFILE.S = OLDFILE.S will rename OLDFILE.S

NOTE: the .S or .O must be entered here, if any, and that quotes should not be used. Also, to prevent unintentional initialization, the N (New) command is not supported.

The address after "SOURCE" and "OBJECT" on the menu give the start address followed by the address of the byte just BEYOND the end. Thus, if "SOURCE:$ØAØØ,$12A5" is indicated, the a SAVE will save the area $ØAØØ to $12A4 inclusive.

NOTE: the command prompt "X" comes up after a Catalog. Execute any disk commands with the X command.



THE EDITOR

Basically there are three modes in the editor: the COMMAND mode, the ADD or INSERT mode, and the EDIT mode. The main one is the COMMAND mode, which has a "colon" (:) as prompt.

Command Mode

For many of the COMMAND mode commands, only the first letter of the command is required, the rest being optional. This manual shows the required command characters in upper case and the optional ones in lower case. In some commands, you must specify a line number, a range of line numbers, or a range list. A line number is just a number. A range is a pair of line numbers separated by a comma. A range list consists of several ranges separated by slashes ("/").

Several commands allow specification of a string. The string must be "delimited" by a non-numeric character other than the slash. Such a delimited string is called a d-string. The usual delimiter is single or double quote marks (' or ").

Line numbers in the editor are provided automatically. You never type them when entering text; only when giving commands. If a line number in a range exceeds the number of the last line, it is automatically adjusted to the last line number. The commands are:

HImem:

HI: (address)

HImem: sets the upper limit for the source file and beginning address for the OBJ file.

NEW

Deletes present source file, resets HIMEM to $8000 and starts fresh.

PORT

PORT (2,4, or 5)

Selects a printer in specified port for output, but does not format output as does the PRTR command.

USER

This does a JSR to the routine whose address is in $32E,$32F. This vector is reset to its default (kernel) value when either an Exit to BASIC or an EXECUTIVE mode command G (run a program) is done. If this is the case, you may use the USER command to send you to the EXEC mode. You can reconnect the SUPPLIED USER routines (CHRGEN 8Ø, XREF, XREF.XL) by doing an A)ØG from the monitor. This avoids having to rerun these utilities.

TABS

TABS <number><, number><, ...> <"tab character">

This sets the tabs for the editor, and has no effect on the assembler listing. Up to nine tabs are possible. The default tab character is a space, but any may be specified. The assembler regards the space as the only acceptable tab character for the separation of labels, opcodes, and operands. If you don't specify the tab character, then the last one used remains. Entering TABS and a RETURN will set all tabs to zero.

LENgth

This gives the length in bytes of the source file, and the number of bytes remaining before MERLIN'S HIMEM (usually $8ØØØ - not BASIC HIMEM).

Where

Where (line number)

This prints in hex the location in memory of the start of the specified line. "Where Ø" (or "WØ") will give the location of the end of source.

MONitor

This exits to the monitor. You may re-enter by "Q" RETURN. This re-establishes the important zero page pointers from a save area inside MERLIN itself. "Q" RETURN will give a correct entry, even if you have messed up the zero page pointers while in the monitor. This facility is designed for experienced programmers, and is not recommended to beginners.

You may re-enter the editor directly with an "R" RETURN command. This re-entry, unlike the others, will use the zero page pointers at $ØC - $11 instead of the ones saved upon exit. Therefore, you must be sure that they have not been altered. For normal usage, however, "Q" RETURN should be used to re-enter MERLIN.

TRuncON

This sets a flag which, during LIST or PRINT, will terminate printing of a line upon finding a space followed by a semicolon. It makes reading of source files easier on the Commodore 4Ø column screen. In the assembler, when used as a pseudo-op, limits printing of the object code to three bytes per line and has no effect on comments.

TRuncOFf

This returns to the default condition of the truncation flag (which also happens automatically upon entry to the editor from the EXEC mode or from the assembler). All source lines when listed or printed will appear normal.

Quit

Exits to EXEC mode.

ASM

This passes control to the assembler, which attempts to assemble the source file. First, however, you are asked if you wish to "update the source". This is to remind you to change the date or identification number in your source file. If you answer "N" then the assembly will proceed. If you answer "Y", you will be presented with the first line in the source containing a "/" and are placed in EDIT mode. When you finish editing this line and hit RETURN, assembly will begin. If you use the STOP edit abort command, however, you will return to the EDITOR command mode, and any I/O hooks you have established, by PORT or PRTR etc., will have been disconnected. This will also happen if there is no line with a "/".

Function key F1

During the second pass of assembly, typing an F1 will toggle the list flag, so that listing will either stop or resume. This will be defeated if a LST opcode occurs in the source, but another F1 will reinstate it.

Delete

Delete (line number) <range> <range list>
Delete (range)
Delete (range list)

This deletes the specified lines. Since, unlike BASIC, the line numbers are fictitious they change with any insertion or deletion. Therefore, you MUST specify the higher range first for the correct lines to be deleted!

Replace

Replace (line number)
Replace (range)

This deletes the line number or range, then places you into INSERT mode at that location.

List

List
List (line number)
List (range)
List (range list)

Lists the source file with added line numbers. Control characters in source are shown in inverse, unless the listing is being sent to a printer or other non-standard outport.

The listing can be aborted by STOP or with "/" key. You may stop the listing by hitting the space bar and then advance a line at a time by hitting the space bar again. Any other key will restart it. This space bar pause also works during assembly and the symbol table print out.

. (period)

Lists starting from the beginning of the last specified range. For example, if you type L1Ø,1ØØ then lines 1Ø to 1ØØ will be listed. If you then use ".", listing will start again at 1Ø and continue until stopped (the end of the range is not remembered).

/

/ <line number>

This continues listing from the last line number listed, or, when a line number is specified, from that line. This listing continues to the end of the file or until it is stopped as in LIST.

Print

Print
Print (line number)
Print (range)
Print (range list)

This is the same as LIST except that line numbers are not added.

PRinTeR

PRinTeR (command)

This command is for sending a listing to a printer with page headers and provision for page boundary skips. The default parameters may be set up by changing the DATA line in the BASIC MERLIN program. The syntax of this is:

PRTR PORT (string) <page header>

If the port number is not 2, 4 or 5, then output is sent to the video screen.

If the page header is omitted, the header will consist of page numbers only.

THE INITIALIZATION STRING MAY NOT BE OMITTED. If your printer does not use a special string, use a null string (""). For example, if the printer is in Port 2 and does not need an initialization string, PRTR 2 "" would be the correct syntax. Commodore 1525 uses a null string, while CTRL-Q is the initilization string for IDS printers.

PRTR Ø (no strings required here) will send output to the screen and allow you to see where the page breaks occur. No output is sent to the printer until a LIST, PRINT, or ASM command is issued.

Find

Find (d-string)
Find (line number) <d-string>
Find (range) <d-string>
Find (range list) <d-string>

This lists those lines containing the specified string. It may be aborted with STOP or "/" key. Since the F7 case toggle works in command mode, you can use it to find or change strings with lower case characters.

Change

Change (d-string d-string)
Change (line number) <d-string d-string>
Change (range) <d-string d-string>
Change (range list) <d-string d-string>

This changes occurrences of the first d-string to the second d-string. The d-string must have the same delimiter with the adjoining ones coalescing. For example, to change occurrences of "speling" to "spelling" throughout the range 2Ø,1ØØ, you would type C2Ø,1ØØ "speling"spelling". If no range is specified, the entire

source file is used.

Before the change operation begins, you are asked whether you want to change "all" or "some". If you select "some" by hitting the "S" key, the editor stops whenever the first string is found and displays the line as it would appear with the change. If you then hit ESCAPE or any control character, the change displayed will not be made. Any other key, such as the space bar, will accept the change. STOP or "/" key will abort the change process.

COPY

COPY (line number) TO (line number)
COPY (range) TO (line number)

This copies the line number or range to just "below" (numerically) the specified number. It does not delete anything.

MOVE

MOVE (line number) TO (line number)
MOVE (range) TO (line number)

This is the same as COPY but after copying, automatically deletes the original range. You always end up with the same lines as before, but in a different order.

Edit

```
Edit
Edit (d-string)
Edit (line number) <d-string>
Edit (range) <d-string>
Edit (range list) <d-string>
```

This presents each line of the line number, range, or range list line by line to be edited and puts you into the EDIT mode. If a d-string is appended, only those lines containing the d-string are presented. See the discussion later in this chapter concerning the EDIT mode commands.

Hex-Dec Conversion

If, in the command mode, you type a decimal number (positive or negative) the hex equivalent is returned. If you type a hex number, prefixed by "$", the decimal equivalent is returned. All commands accept hex numbers, which is mainly convenient for the HImem: and SYM commands.

TEXT

This converts ALL spaces in a source file to inverse spaces. The purpose is for use on "text" files so that it is not necessary to remember to zero the tabs before printing such a file. This conversion has no effect on anything except the editor's tabulation.

FIX

This undoes the effect of TEXT. It also does a number of technical housekeeping chores. It is recommended that the command FIX be used on all files from external sources, after which the file should be saved.

NOTE: the TEXT and FIX routines are written in SWEET 16 and are somewhat slow. Several minutes may be needed for their execution on large files. FIX or an EDIT will truncate any lines longer than 255 characters.

SYM

SYM (address)

MERLIN normally places the symbol table in the RAM behind the KERNAL ROM ($E000-$FD00). This space is quite adequate for all but gigantic programs. In case this space is used up, the SYM command gives you a means to direct the assembler to continue the symbol table in another area. If you type SYM $9000, for example, and assemble the program, when and if the symbol table uses up its normal space, it will be continued at $9000 until it reaches BASIC HIMEM. It must be noted that the SYM command will be cancelled by a HIMEM command or by exit to EXEC mode and re-entry (set HIMEM before setting up a SYM address).

The SYM address must be above HIMEM and below BASIC HIMEM. If the symbol table grows beyond the allotted space, you will get a memory error during the first pass of assembly.

VIDeo

This command toggles the 8Ø column mode, provided a Data 2Ø Videopak 8Ø cartridge is installed. Note that the 8Ø column mode uses up 4K of source space ($9ØØØ - $9FFF). For long sources it may be necessary to use the 8Ø column mode only for editing and then switch to the 4Ø column mode for assembly. This is possible since switching to the 4Ø column mode automatically retreives the 4K of space. Note also that MERLIN does NOT support the "Videopak 8Ø" 4Ø column mode. For technical reasons, the VID command sends you back to the EXEC mode after the switch.

FW (Find Word)

```
FW (d-string)
FW (line number) <d-string>
FW (range) <d-string>
FW (range list) <d-string>
```

This is an alternative to the FIND command. It will find the specified word only if it is surrounded, in source, by non-alphanumeric characters. Therefore, FW"CAT" will find:

```
CAT
CAT-1
(CAT,X)
```

but will not find CATALOG or SCAT.

CW (Change word)

```
CW (d-string d-string)
CW (line number) <d-string d-string>
CW (range) <d-string d-string>
CW (range list) <d-string d-string>
```

This works similar to the CHANGE command with the added features as described under FW.

EW (Edit word)

EW (d-string)
EW (line number) <d-string>
EW (range) <d-string>
EW (range list) <d-string>

This is to EDIT as FW is to FIND.

NOTE ON DELIMITED STRINGS: For the commands involving delimited strings (a d-string), the character "^" acts as a wild card. Therefore, F"Jon^s" will find both "Jones" and "Jonas".

VAL

VAL "expression"

This will return the value of the expression as the assembler would compute it.

Examples:

VAL "LABEL"	gives the address (or value) of LABEL for the last assembly done or "unknown label" if not found.
VAL "$1ØØØ/2"	returns $Ø8ØØ
VAL "%1ØØØ"	returns $ØØØ8
VAL !"a"-"Ø"!	returns $ØØ11

ERR

This reads the error channel on the current disk drive and prints out the error information. This should be used if some disk operation makes the drive light blink but does not produce an error message. There are certain disk errors that the operating systems does not signal back to the calling program. This provides a means for checking them.

Add/Insert Mode

The ADD and INSERT modes in the editor act as if you are in the edit mode, except that CTRL-R will do nothing, and the exit from ADD mode acts as described. Hitting RETURN, for example, will accept the entire line as shown on the screen.

add

This places you in the ADD mode. You may enter lower case text (useful for comments if you have a lower case adapter) by hitting F7. This acts as a case toggle, so another F7 returns you to UPPERCASE mode. The shift key inverts the current case. Shift lock should not ordinarily need to be used. To exit from ADD mode, hit RETURN as the FIRST character of a line. You may also exit the ADD mode by CTRL-X or STOP which also cancels the current line.

You may enter an EMPTY line by typing a space and then RETURN. This will not enter the space into text, it only bypasses the exit. The editor automatically removes extra spaces at the end of lines.

Insert

Insert <line number>

This allows you to enter text just below (numerically) the specified line. Otherwise, it functions the same as ADD command (above).

Function key F7 (or Control-L)

Toggles the current case. If you are in upper case, F7 will place you in lower, and vice versa. Upper case is defaulted to when entering each new line.

To change the case of a word, type F7, then copy over the word using the right hand cursor control key.

Edit Mode

After typing E in the editor, you are placed in EDIT mode. The first line of the range you have specified is placed on the screen with the cursor on its first character. The line is tabbed as it is in listing, and the cursor will jump across the tabs as you move it with the arrow keys. When you are through editing, hit RETURN. The line will be accepted as it appears on the screen, no matter where the cursor is when you hit RETURN.

The cursor keys do not work as their markings suggest in edit mode. The up/down arrow key works as a move cursor left key and the left/right arrow key works as a move cursor right key.

Edit Mode Commands

Control-I or the Insert Key (insert)

Begins insertion of characters. This is terminated by any control character except the CTRL-L case toggle, such as the arrows or RETURN.

Control-D (delete)

Deletes the character under the cursor. It can also be referred to as a backwards delete.

Control-F (find)

Finds the next occurence of the character typed after the CTRL-F. This is recursive.

Control-O (insert special)

Functions as CTRL-I, except it inserts any control character (including the command characters such as CTRL-A).

Control-P (do ***'s)

If entered as the first character of a line gives 32 *'s. If entered as any other character of the line, gives 30 spaces bordered by *'s.

Stop Key (cancel)

Aborts EDIT mode and returns to the editor's command mode. The current line being edited will retain its original form.

Control-B (go to line begin)

Places the cursor at the beginning of the line.

Control-N (go to line end)

Places the cursor one space past the end of the line.

Control-R (restore line)

Returns the line to its original form. (Not available in ADD and INSERT modes.)

Control-A (delete line right)

Deletes the part of the line following the cursor and terminates editing.

Return (RETURN key)

Accepts the line as it appears on the screen and fetches the next line to be edited, or goes to the command mode if the specified range has been completed.

The editor automatically replaces spaces in comments and ASCII strings with inverse spaces. When listing, it converts them back, so you never notice this. Its purpose is to avoid inappropriate tabbing of comments and ASCII strings.

In the case of ASCII strings, this is only done when the delimiter is a quote (") or a single quote (´). You can, however, accomplish the same thing by editing the line, replacing the first delimiter with a quote, hitting RETURN, then editing again and changing the delimiter back to the desired one.

In a line such as LDA #´ ´, you can prevent the extra tabbing by entering the line with a space before the first quote (LDA # ´ ´), then typing control-N and then using the cursor control to delete the extra space.

THE ASSEMBLER

This section of the documentation will not attempt to teach you assembly language. It will only explain the syntax you are expected to use in your source files, and document the features that are available to you in the assembler.

Number Format

The assembler accepts decimal, hexadecimal, and binary numerical data. Hex numbers must be preceded by "$" and binary numbers by "%", thus the following four instructions are all equivalent:

```
LDA #100    LDA #$64    LDA #%11001100    LDA #%01100100
```

As indicated, leading zeros are ignored. The "#" here stands for "number" or "data", and the effect of these instructions is to load the accumulator with the number (decimal) 100.

A number not preceded by "#" is interpreted as an address. Therefore:

```
LDA 1000    LDA $3E8    LDA %1111101000
```

are equivalent ways of loading the accumulator with the byte that resides in memory location $3E8.

Use the number format that is appropriate for clarity. For example, the data table:

```
DA    $1
DA    $A
DA    $64
DA    $3E8
DA    $271Ø
```

is a good deal more mysterious that its decimal equivalent:

```
DA    1
DA    1Ø
DA    1ØØ
DA    1ØØØ
DA    1ØØØØ
```

Source Code Format

A line of source code typically looks like:

```
LABEL OPCODE OPERAND ;COMMENT
```

A line containing only a comment must begin with "*". Comment lines starting with ";" are accepted and tabbed to the comment field. The assembler will accept an empty line in the source code and will treat it just as a SKP 1 instruction (see the section on pseudo opcodes), except the line number will be printed.

The number of spaces separating the fields is not important, except for the editor's listing, which expects just one space.

The maximum allowable LABEL length is 13 characters, but more than 8 will produce messy assembly listings. A label must begin with a character at least as large, in ASCII value, as the colon, and may not contain any characters less, in ASCII value, than the number zero.

A line may contain a label by itself. This is equivalent to equating the label to the current value of the address counter.

The assembler examines only the first 3 characters of the OPCODE. For example, you can use PAGE instead of PAG. The assembler listing will truncate the opcode to seven letters and will not look well with one longer than four unless there is no operand.

The maximum allowable combined OPERAND + COMMENT length is 64 characters. You will get an error if you use more than this. A comment line by itself is also limited to 64 characters.

Expressions

To make clear the syntax accepted and/or required by the assembler, we must define what is meant by an "expression". Expressions are built up from "primitive expressions" by use of arithmetic and logical operations. The primitive expressions are:

1. A label.
2. A number (either decimal, $hex, or %binary).
3. Any ASCII character preceded or enclosed by quotes or single quotes.
4. The character * (standing for the present address).

All number formats accept 16-bit data and leading zeros are never required. In case 3, the "value" of the primitive expression is just the ASCII value of the character in Commodore U/L case format. The high-bit will be on if a quote (") is used, and off if a single quote (') is used.

The assembler supports the four arithmetic operations: +, -, / (integer division), and * (multiplication). It also supports the three logical operations: ! (Exclusive OR), . (OR), and & (AND).

Some examples of legal expressions are:

```
LABEL1-LABEL2          (LABEL2 minus LABEL1)
2*LABEL+$231           (2 times LABEL plus hex 231)
1234+%10111        (1234 plus binary 10111)
"K"-"A"+1              (ASCII "K" minus ASCII "A" plus 1)
"0"!LABEL          (ASCII "0" EOR LABEL)
LABEL&$7F              (LABEL AND hex 7F)
*-2                    (present address minus 2)
LABEL.%10000000    (LABEL OR binary 10000000)
```

Parentheses have another meaning and are not allowed in expressions. All arithmetic and logical operations are done from left to right (2+3*5 would assemble as 25 and not 17).

Immediate Data

For those opcodes such as LDA, CMP, etc., which accept immediate data (numbers as opposed to addresses) the immediate mode is signalled by preceding the expression with "#". An example is LDX #3. In addition:

#<expression	produces the low byte of the expression
#>expression	produces the high byte of the expression
#expression	also gives the low byte (the 6510 does not accept 2-byte DATA)
#/expression	is optional syntax for the high byte of the expression

The ability of the assembler to evaluate expressions such as LAB2-LAB1-1 is very useful for the following type of code:

```
COMPARE     LDX   #EODATA-DATA-1
LOOP        CMP   DATA,X
            BEQ   FOUND       ;found
            DEX
            BPL   LOOP
            JMP   REJECT      ;not found
DATA        HEX   E3BC3498
EODATA      EQU   *
```

With this type of code, you can add or delete some of the DATA and the value which is loaded into the X index for the comparison loop will be automatically adjusted.

Addressing Modes (for 651Ø Opcodes)

The assembler accepts all the 65Ø2 opcodes with standard mnemonics. It also accepts BLT (branch if less than) and BGE (branch if greater or equal) as synonymous to BCC and BCS.

There are 12 addressing modes on the 651Ø. The appropriate MERLIN syntax for these are:

	Syntax	Example
Implied	OPCODE	CLC
Accumulator	OPCODE	ROR
Immediate (data)	OPCODE #expr	ADC #$F8
		CMP #"M"
		LDX #>LABEL1-LABEL2-1
Zero page (address)	OPCODE expr	ROL 6
Indexed X	OPCODE expr,X	LDA $EØ,X
Indexed Y	OPCODE expr,Y	STX LAB,Y
Absolute (address)	OPCODE expr	BIT $3ØØ
Indexed X	OPCODE expr,X	STA $4ØØØ,X
Indexed Y	OPCODE expr,y	SBC LABEL-1,Y
Indirect	JMP (expr)	JMP ($3F2)
Preindexed X	OPCODE (expr,X)	LDA (6,X)
Postindexed Y	OPCODE (expr),Y	STA ($FE),Y

NOTE: There is no difference in syntax for zero page and absolute modes. The assembler automatically uses zero page mode when appropriate. MERLIN provides the ability to FORCE non-zero page addressing. The way to do this is to add anything to the end of the opcode. Example:

```
LDA  $1Ø  assembles as zero page (2 bytes) while,
LDA: $1Ø  assembles as non-zero page (3 bytes).
```

Also, in the indexed indirect modes, only a zero page expression is allowed, and the assembler will give an error message if the "expr" does not evaluate to a zero page address.

NOTE: The "accumulator mode" does not require an operand (the letter "A"). Some assemblers perversely require you to put an "A" in the operand for this mode.

The assembler will decide the legality of the addressing mode for any given opcode.

Pseudo Opcodes - Directives

EQU (=) (EQUate)

```
Label  EQU  expression
Label  =    expression    (alternate syntax)
```

Used to define the value of a LABEL, usually an exterior address or an often used constant for which a meaningful name is desired. It is recommended that these all be located at the beginning of the program. The assembler will not permit an "equate" to a zero page number after the label equated has been used, since bad code could result from such a situation (also see "Variables").

ORG (set ORIGin)

```
ORG expression
```

Establishes the address at which the program is designed to run. It defaults to the present value of HIMEM: ($8000 by default). Ordinarily there will be only one ORG and it will be at the start of the program. If more than one ORG is used, the first one establishes the LOAD address. This can be used to create an object file that would load to one address though it may be designed to run at another address.

You cannot use ORG *-1 to back up the object pointers as is done in some assemblers. This must be done instead by DS -1.

PUT (PUT a text file in assembly)

PUT filename

"PUT filename", (drive parameter accepted in standard syntax) reads the named file and "inserts" it at the location of the opcode.

NOTE: "Insert" refers to the effect on assembly of the location of the source. The file itself is actually placed just following the main source. Text files are required by this facility in order to insure memory protection. A memory error will occur if a PUT file goes beyond HIMEM:. These files are in memory only one at a time, so a very large program can be assembled using the PUT facility.

There are two restrictions on a PUT file. First, there cannot be MACRO definitions inside a PUT file, they must be in the main source. Second, a PUT file may not call another PUT file with the PUT opcode. Of course, linking can be simulated by having the "main program" just contain the macro definitions and call, in turn, all the others with the PUT opcode.

Any variables (such as]LABEL) may be used as "local" variables. The usual local variables]1 through]8 may be set up for this purpose using the VAR opcode.

The PUT facility provides a simple way to incorporate much used subroutines, such as PRINT or PRDEC, in a program.

VAR (setup VARiables)

VAR expr;expr;expr...

This is just a convenient way to equate the variables]1 thru]8. "VAR 3;$42;LABEL" will set]1 = 3,]2 = $42, and]3 = LABEL. This is designed for use just prior to a PUT. If a PUT file uses]1 -]8, except in >>> lines

for calling macros, there MUST be a previous declaration of these.

SAV (SAVe object code)

SAV filename

"SAVE filename", (drive parameter accepted) will save the current object code under the specified name. This acts exactly as does the EXEC mode object saving command, but it can be done several times during assembly.

This pseudo-opcode provides a means of saving portions of a program having more than one ORG. It also enables the assembly of extremely large files. After a save, the object address is reset.

The SAVE command sets the address of the saved file to its correct value. For example, if your program contains three SAV commands, then it will be saved in three pieces. When LOADed later with the LOAD "FILE", 8,1 syntax, they will go to the correct locations, the third following the second and that following the first.

Together, the PUT and SAV opcodes make it possible to assemble extremely large files.

DSK (assemble directly to DiSK)

DSK filename

"DSK filename" will direct the assembler to assemble the following code directly to disk. If DSK is already in effect, the old file will be closed and the new one begun. This is useful primarily for extremely large files.

END (END of source file)

```
 END
```

This rarely used or needed pseudo opcode instructs the assembler to ignore the rest of the source. Labels occurring after END will not be recognized.

DUM (DUMmy section)

```
 DUM expression
```

This starts a section of code that will be examined for value of labels but will produce no object code. The expression must give the desired ORG of this section. It is possible to re-ORG such a section using another DUMMY opcode or using ORG. It is legal to use DS opcodes in dummy sections but, since the address is not printed for a DS opcode, it is preferable to use other forms (DA, DFB, etc). Note that, although no object code is produced from a dummy section, the text output of the assembler will appear as if code is being produced.

DEND (Dummy END)

```
 DEND
```

This ends a dummy section and re-establishes the ORG address to the value it had upon entry to the dummy section.

Formatting

LST ON/OFF (LiSTing control)

LST ON or OFF

This controls whether the assembly listing is to be sent to the screen (or other output device) or not. You may, for example, use this to send only a portion of the assembly listing to your printer. Any number of LST instructions may be in the source. If the LST condition is OFF at the end of assembly, then the symbol table will not be printed.

The assembler actually only checks the third character of the operand to see whether or not it is a space. Therefore, LST FRED will have the same effect as LST OFF. The LST directive will have no effect on the actual generation of object code. If the LST condition is OFF, the object code will be generated much faster, but this is recommended only for debugged programs.

NOTE: Function key F1 from the keyboard toggles this flag during the second pass.

EXP ON/OFF (macro EXPand control)

EXP ON or OFF

EXP ON will print an entire macro during the assembly. The OFF condition will print only the >>> pseudo-op. EXP defaults to ON. This has no effect on the object coded generated.

PAU (PAUse)

PAU

On the second pass this causes assembly to pause until a key is hit. This can also be done from the keyboard by hitting the space bar.

PAG (new PAGe)

PAG

This sends a formfeed ($ØC) to the printer. It has no effect on the screen listing even when using an 8Ø-column cartridge. Note that the Commodore 1525 printer does not recognize a form feed. If PRTR is in effect, however, this character will be intercepted and acted upon properly.

AST (send a line of ASTerisks)

AST expression

This sends a number of asterisks (*) to the listing equal to the value of the expression. The number format is the usual one, so that AST 1Ø will send (decimal) 1Ø asterisks, for example. The number is treated modulo 256 with Ø being 256 asterisks!

SKP (SKiP lines)

SKP expression

This sends "expression" number of carriage returns to the listing. The number format is the same as in AST.

TR ON/OFF (TRuncate control)

TR ON or OFF

TR ON limits object code printout to three bytes per source line, even if the line generates more than three. TR OFF resets it to print all the object bytes.

Strings

The opcodes in this section also accept hex data after the string. Any of the following syntaxes are acceptable:

```
ASC "string"878DØØ
ASC "string",878DØØ
ASC "string",87,8D,ØØ
```

TXT (define TeXT)

TXT d-string

This puts a delimited Commodore ASCII string into the object code. The only restriction on the delimiter is that it does not occur in the string itself. Different delimiters have different effects. Any delimiter less than (in ASCII code) the single quote (') will produce a string with the high-bits on, otherwise the high-bits will be off. For example, the delimiters !"#$%& will produce a string in "negative" ASCII, and the delimiters '()+? will produce one in "positive" ASCII. Usually the quote (") and single quote (') are the delimiters of choice, but other delimiters provide the means of inserting a string containing the quote or single quote as part of the string.

DCI (Dextral Character Inverted)

DCI d-string

This is the same as TXT except that the string is put into memory with the last character having the opposite high bit as the others. All choices for delimiters otherwise have the same effect as TXT.

ASC (define ASCii text)

ASC d-string

This is the same as TXT except that it uses standard ASCII annotation.

ASI (define AScii Inverted text)

ASI d-string

This is the same as ASC except that the string is put into memory with the last character having the opposite high bit to the others. ASI is to ASC as DCI is to TXT.

REV (define REVerse text)

REV d-string

This puts the d-string in memory backwards. Example:

REV "RACECAR" gives "RACECAR"
(maybe that wasn't such a good example)

REV "COMMODORE" gives "ERODOMMOC"

Delimiter choices are the same as with TXT. HEX data cannot be added after the string terminator.

STR (define text with length byte for STRings)

STR d-string

This is the same as TXT but places a length byte preceding the string. The syntax will accept hex data after the string only, but the length will not include any hex bytes.

In summary:

Command	Input	Annotation	Assembles as (hex)
TXT	'Abc'	Commodore ASCII	61 42 43
DCI	'Abc'	Commodore ASCII	61 42 C3
REV	'Abc'	Commodore ASCII	43 42 61
ASC	'Abc'	Standard ASCII	41 62 63
ASI	'Abc'	Standard ASCII	41 62 E3

Data and Allocation

DA (Define Address)

DA expression

This stores the two-byte value of the operand, usually an address, in the object code, low-byte first. For example, DA $FDFØ will generate FØ FD. Also accepts multiple data (e.g. DA 1,1Ø,1ØØ)

DDB (Define Double Byte)

DDB expression

As above, but places high-byte first. Also accepts multiple data (e.g. DDB 1,1Ø,1ØØ).

DFB (DeFine Bytes)

DFB expression

This puts the bytes specified by the operand into the object code. It accepts several bytes of data, which must be separated by commas and contain no spaces. The standard number format is used and arithmetic is done as usual.

The "#" symbol is acceptable but ignored, as is "<". The ">" symbol may be used to specify the high-byte of the label, otherwise the low-byte is always taken. The ">" symbol should appear only as the first character of an expression or immediately after #. That is, the instruction DFB >LAB1-LAB2 will produce the high-byte of the value of LAB1-LAB2.

For example:

```
DFB $34,1ØØ,LAB1-LAB2,%1ØI1,>LAB1-LAB2
```

is a properly formatted DFB statement which will generate the object code (hex):

```
34 64 DE ØB Ø9
```

assuming that LAB1=$81A2 and LAB2=$77C4.

HEX (define HEX data)

HEX hex-data

This is an alternative to DFB which allows convenient insertion of hex data. Unlike all other cases, the "$" is not required or accepted here. The operand should consist of hex numbers having two hex digits (e.g. ØF, not F). They may be separated by commas or may be adjacent. An error message will be generated if the operand contains an odd number of digits or ends in a comma, or, as in all cases, contains more than 64 characters.

DS (Define Storage)

DS expression

This reserves space for storage data. It zeros out this space if the expression is positive. DS 1Ø, for example, will set aside 1Ø bytes for storage. Because DS adjusts the object code pointer, and instruction like DS -1 can be used to back up the object and address pointers one byte.

KBD (define label from KeyBoarD)

label KBD <d-string>

This allows a label to be equated from the keyboard during assembly. Any expression may be inputted, including expressions referencing previously defined labels, however a BAD INPUT error will occur if the input cannot be evaluated. If the operand is empty, then the message "Give value for: ", followed by the the label, is printed on the screen. If the operand contains a delimited string then this string is printed instead. See MACRO LIBRARY and KEYMAC for examples of this.

LUP (begin a loop)

```
 LUP expression   (Loop)
 --^              (end of LUP)
```

The LUP pseudo-opcode is used to repeat portions of source between the LUP and the --^ "expression" number of times. An example of the syntax for this is:

```
LUP 4
ASL
--^
```

Which will assemble as:

```
ASL
ASL
ASL
ASL
```

and will show that way in the assembly listing, with repeated line numbers.

Perhaps the major use of this is for table building. As an example:

```
]A  = Ø
    LUP $FF
]A  = ]A+1
    DFB ]A
    --^
```

will assemble the table 1, 2, 3, ..,$FF.

The maximum LUP value is $8ØØØ and the LUP opcode will simply be ignored if you try to use more than this.

CHK (place CHecKsum in object code)

```
 CHK
```

This places a checksum byte into object code at the location of the CHK opcode (usually at the end of the program).

ERR (force ERRor)

```
 ERR expression
```

"ERR expression" will force an error if the expression has a non-zero value and the message BREAK IN LINE ??? will be printed.

This may be used to ensure your program does not exceed, for example, $95FF by adding the final line:

```
    ERR  *-1/$96ØØ
```

NOTE: This would only alert you that the program is too long, and will not prevent writing above $96ØØ during assembly, but there can be no harm in this. The error occurs only on the second pass of the assembly and does not abort the assembly.

Another available syntax is:

```
ERR  ($3ØØ)-$4C
```

which will produce an error on the first pass and abort assembly if location $3ØØ does not contain the value $4C.

USR (USeR definable opcode)

USR optional expressions

This is a user definable pseudo opcode. It does a JSR $335. To set up your routine you should RUN it with the EXEC mode "G" command. Your routine can start at address $334 with an RTS at that location, so that, effectively, the "G" command simply loads the remainder of the program to $335. The following flags and entry points may be used by your routine:

```
USRADS    = $335   ;start of your routine
PUTBYTE   = $B726
EVAL      = $B729
PASSNUM   = $4     ;contains assembly pass number
ERRCNT    = $1F    ;error count
VALUE     = $55,56 ;value returned by EVAL
OPNDLEN   = $2B    ;contains combined length of
                   ;operand and comment
NOTFOUND  = $3D    ;see discussion of EVAL
WORKSP    = $98Ø   ;contains the operand and
                   ;comment in positive ASCII
```

Your routine will be called by the USR opcode with A=Ø, Y=Ø and carry set. To direct the assembler to put a byte in the object code, you should JSR PUTBYTE with the byte in A.

PUTBYTE will preserve Y but will scramble A and X. It returns with the zero flag clear (so that BNE always branches). On the first pass, PUTBYTE adjusts the object and address pointers, so that the contents of the registers are not important. You MUST call PUTBYTE the SAME NUMBER OF TIMES on each pass or the pointers will not be kept correctly and the assembly of other parts of the program will be incorrect!

If your routine needs to evaluate the operand, or part of it, you can do this by a JSR EVAL. The X register must point to the first character of the portion of the operand you wish to evaluate (put X=Ø to evaluate the expression at the start of the operand). On return from EVAL, X will point to the character following the evaluated expression. The Y register will be Ø, 1, or 2 depending on whether this character is a right parenthesis, a space or a comma.

Any other character not allowed in an expression will cause assembly to abort with a BAD OPERAND error. If some label in the expression is not recognized then location NOTFOUND will be non-zero. On the second pass, however, you will get an UNKNOWN LABEL error and the rest of your routine will be ignored. On return from EVAL, the computed value of the expression will be in location VALUE and VALUE+1, lowbyte first. On the first pass this value will be insignificant if NOTFOUND is non-zero.

Your routine may use zero page locations $6Ø-$6F for scratch work but other zero page locations should not be altered. Upon return from your routine (RTS), the USR line will be printed (on the second pass).

When you use the USR opcode in a source file, it is wise to include some sort of check (in source) that the required routine is in memory. If, for example, your routine contains the byte $3 at location $34Ø then:

```
ERR ($34Ø)-$3
```

will test that byte and abort assembly if it is not

there. Similarly, if you know that the required routine should assemble exactly two bytes of data, then you can (roughly) check for it by the following code:

```
LABEL    USR  OPERAND
         ERR  *-LABEL-2
```

This will force an error on the second pass if USR does not produce exactly two object bytes.

It is possible to use USR for several different routines in the same source. For example, your routine could check the first operand expression for an index to the desired routine and act accordingly. Thus "USR 1, whatever" would branch to the first routine, "USR 2,stuff" to the second, etc.

Conditionals

DO (DO if true)

DO expression

This, together with ELSE and FIN are the conditional assembly PSEUDO-OPS. If the operand evaluates to ZERO, then the assembler will stop generating object code (until it sees another conditional). Except for macro names, it will not recognize any labels in such an area of code. If the operand evaluates to a non-zero number, then assembly will proceed as usual. This is very useful for MACROS.

It is also useful for sources designed to generate slightly different code for different situations. For example, if you are designing a program to go to on a ROM chip, you would want one version for the ROM, and another, with small differences to create a RAM version for debugging purposes.

By using conditional assembly, modification of such programs becomes much simpler, since you do not have to make the modification in two separate versions of the source code. Every DO should be terminated somewhere later by a FIN and each FIN should be preceded by a DO. An 'ELSE should occur only inside such a DO/FIN structure. DO/FIN structures may be nested up to eight deep (possibly with some ELSE's between). If DO condition is off (value Ø), then assembly will not resume until its corresponding FIN is encountered, or an ELSE at this level occurs. Nested DO/FIN structures are valuable for putting conditionals in MACROS.

ELSE (ELSE do this)

ELSE

This inverts the assembly condition (ON becomes OFF and OFF becomes ON) for the last DO.

IF (IF so then do)

IF char,]var (IF char is the first character of]var)

This checks to see if char is the leading character of the replacement string]var. Position is important: the assembler checks the first and third characters of the operand for a match. If a match is found then the following code will be assembled. As with DO, this must be terminated with a FIN, with optional ELSEs between. The comma is not examined, so any character may be used there. For example, IF (=]1.

FIN (FINish conditional)

FIN

This cancels the last DO or IF and continues assembly with the next highest level of conditional assembly, or ON if the FIN concluded the last (outer) DO or IF.

Example of the use of conditional assembly:

```
MOV     MAC
        LDA  ]1
        STA  ]2
        <<<

MOVD    MAC
        MOV  ]1;]2
        IF   (,]1           ;Syntax MOVD (ADR1),Y;????
        INY
        IF   (,]2           ;  MOVD (ADR1),Y;(ADR2),Y
        MOV  ]1;]2
        ELSE                ;  MOVD (ADR1),Y;ADR2
        MOV  ]1;]2+1
        FIN
        ELSE
        IF   (,]2           ;Syntax MOVD ????;(ADR2),Y
        INY
        IF   #,]1           ;  MOVD #ADR1;(ADR2),Y
        MOV  ]1/$1ØØ;]?
        ELSE                ;  MOVD ADR1;(ADR2),Y
        MOV  ]1+1;]2
        FIN
        ELSE                ;Syntax MOVD ????;ADR2
        IF   #,]1           ;  MOVD #ADR1;ADR2
        MOV  ]1/$1ØØ;]2+1
        ELSE                ;  MOVD ADR1;ADR2
        MOV  ]1+1;]2+1
        FIN                 ;MUST close ALL
        FIN                 ;conditionals, Count DOs
        FIN                 ;& IFs, deduct FINs.  Must
        <<<                 ;yield zero at end.
```

* Call syntaxes supported by MOVD:

```
        MOVD ADR1;ADR2
        MOVD (ADR1),Y;ADR2
        MOVD ADR1;(ADR2),Y
        MOVD (ADR1),Y;(ADR2),Y
        MOVD #ADR1;ADR2
        MOVD #ADR1;(ADR2),Y
```

Macros

MAC (begin MACro definition)

```
label MAC
```

This signals the start of a MACRO definition. It must be labeled with the macro name. The name you use is then reserved and cannot be referenced by things other than the >>> pseudo-op (things like DA NAME will not be accepted if NAME is the label on MAC). See the section on MACROS for details of the usage of macros.

<<< (end of macro)

```
<<<
```

This signals the end of the definition of a macro. It may be labeled and used for branches to the end of a macro, or one of its copies.

>>> (include macro)

```
>>> macro-name
```

This instructs the assembler to assemble a copy of the named macro at the present location. See the section on MACROS. It may be labeled.

Variables

Labels beginning with "]" are regarded as VARIABLES. They can be redefined as often as you wish. The designed purpose of variables is for use in MACROS, but they are not confined to that use.

Forward reference to a variable is impossible (with correct results) but the assembler will assign some value to it. That is, a variable should be defined before it is used.

It is possible to use variables for backwards branching, using the same label at numerous places in the source. For example:

```
 1          LDY #Ø
 2 ]JLOOP   LDA TABLE,Y
 3          BEQ NOGOOD
 4          JSR DOIT
 5          INY
 6          BNE ]JLOOP       ;BRANCH TO LINE 2
 7 NOGOOD   LDX #-1
 8 ]JLOOP   INX
 9          STA DATA,X
1Ø          LDA TBL2,X
11          BNE ]JLOOP       ;BRANCH TO LINE 8
```

MACROS

Defining a Macro

A macro definition begins with the line:

```
 Name  MAC     (no operand)
```

with Name in the label field. Its definition is terminated by the pseudo-op <<<. The label Name cannot be referenced by anything other than >>> Name.

The conditionals DO, ELSE and FIN may be used inside a macro.

Labels inside macros are updated each time >>> is encountered.

Error messages generated by errors in macros usually abort assembly, because of possibly harmful effects. Such messages will usually indicate the line number of a >>> rather than the line inside the macro where the error occurs.

Nested Macros

Macros may be nested to a depth of 15.

Here is an example of a nested macro in which the definition itself is nested. (This can only be done when both definitions end at the same place.)

```
TRDB MAC
     >>> TR.]1+1;]2+1
TR   MAC
     LDA ]1
     STA ]2
     <<<
```

In this example >>> TR.LOC;DEST will assemble as:

```
     LDA LOC
     STA DEST
```

and >>> TRDB.LOC;DEST will assemble as:

```
        LDA LOC+1
        STA DEST+1
        LDA LOC
        STA DEST
```

A more common form of nesting is illustrated by these two macro definitions (where CH = $D3):

```
POKE MAC
     LDA #]2
     STA ]1
     <<<

HTAB MAC
     >>> POKE.CH;]1
     <<<
```

MACRO names may also be put in the opcode column, rather than the >>>, with the following restriction:

> The first three characters of a name must not coincide with any regular opcode. It is acceptable if two MACRO names have the same first three characters.
>
> Exception: The fourth character of an opcode or MACRO name modifies the recognition of the name if it is a 'D'. Thus a MACRO name INCD will not conflict with the opcode INC.

Note that the >>> syntax is still available and is not subject to this restriction. The seven character limitation has been removed.

Special Variables

Eight variables, named]1 through]8, are predefined and are designed for convenience in MACROS. These are used in a >>> statement. The instruction:

```
>>> NAME expr1;expr2;expr3...
```

will assign the value of expr1 to the variable]1, that of expr2 to]2, and so on. An example of this usage is:

```
TEMP    EQU     $1Ø
SWAP    MAC
        LDA     ]1
        STA     ]3
        LDA     ]2
        STA     ]1
        LDA     ]3
        STA     ]2
        <<<
        >>>     SWAP $6;$7;TEMP
        >>>     SWAP $1ØØØ;$6;TEMP
```

This program segment swaps the contents of location $6 with that of $7, using TEMP as a scratch depository, then swaps the contents of $6 with that of $1ØØØ.

If, as above, some of the special variables are used in the MACRO definition, then values for them must be specified in the >>> statement. In the assembly listing, the special variables will be replaced by their corresponding expressions.

The number of values must match the number of variables in the macro definition. A BAD OPERAND error will be generated if the number of values is less than the number of variables. No error message will be generated, however, if there are more values than variables.

The assembler will accept some other characters in place of the space between the macro name and the expressions in a >>> statement. For example, you may use any of these characters:

. / , - (

The semicolons are required, however, and no extra spaces are allowed.

Macros will accept literal data. Thus the assembler will accept the following type of macro call:

```
MUV  MAC
     LDA  ]1
     STA  ]2
     <<<
     >>>  MUV.(PNTR),Y;DEST
     >>>  MUV.#3;FLAG,X
```

It will also accept:

```
PRINT  MAC
       JSR  SENDMSG
       TXT  ]1
       BRK
       <<<
       >>>  PRINT.!"quote"!
       >>>  PRINT.'This is an example'
       >>>  PRINT."So's this, understand?"
```

LIMITATION: If such strings contain spaces or semicolons, they MUST be delimited by quotes (single or double). Also, literals such as >>> WHAT."A" must have the final delimiter. (This is only true in macro calls or VAR statements, but it is good practice in all cases.)

The Macro Library

A macro library with two example macro programs is included in source file form on this diskette. The purpose of the library is to provide some guidance to the newcomer to macros and how they can be used within an assembly program.

NOTE: All macros are defined at the beginning of the source file, then each example program places the macros where they are needed. Conditionals are used to determine which example program is to be assembled. The KBD opcode allows the user to make this selection from the keyboard during assembly.

TECHNICAL INFORMATION

The SOURCE is placed at $AØ1 when loaded, regardless of its original address.

The important pointers are:

```
START OF SOURCE in  $C,$D   (always set $AØ1)
HIMEM           in  $E,$F   (defaults to $8ØØØ)
END OF SOURCE   in  $1Ø,$11
```

When you exit to BASIC or to the monitor, these pointers are saved at $AØØ8-$AØØD. They are restored upon re-entry to MERLIN.

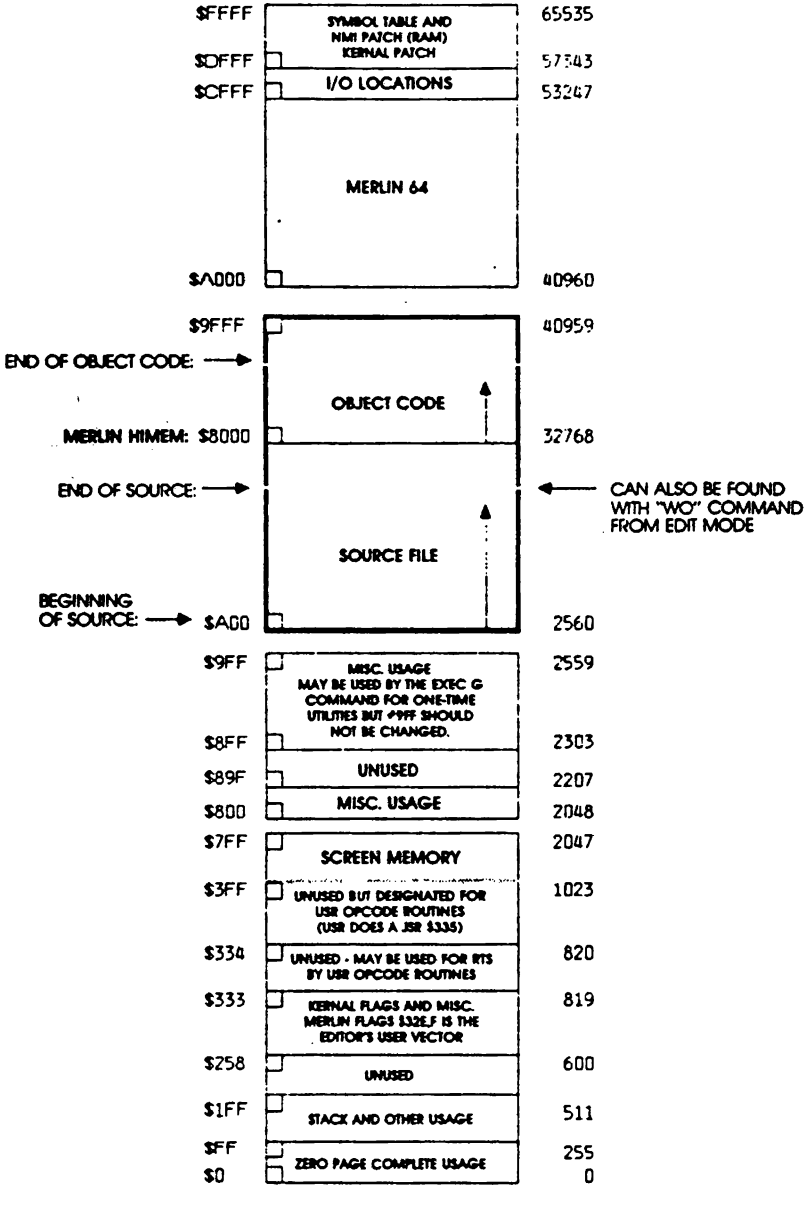
General Information

To re-enter MERLIN after going to BASIC, use the SYS 52ØØØ command.

Memory organization, for ordinary sized files is of no concern to the user, but it is important to understand certain constraints for the handling of large files. MERLIN'S HIMEM (which defaults to $8ØØØ) is an upper limit to the source file. It is also an upper limit for PUT files. If a memory error occurs during assembly, indicating a PUT line, it means the PUT file exceeded HIMEM and that HIMEM will have to be increased.

The default ORG address equals the present value of HIMEM. If during assembly, the object code exceeds BASIC HIMEM (or the SYM address, if one has been specified) then the code will not be written to memory, but assembly will appear to proceed and its output sent to the screen or printer. The only clue that this has happened, if not intentional, is that the object SAVE command is disabled in this event. Therefore, if a listing for a very long file is desired, without actually creating code, the user can assemble over MERLIN itself.

MERLIN 64 MEMORY MAP



Symbol Table

The symbol table is printed after assembly unless LST OFF has been invoked. It comes first in alphabetical order and then in numerical order. The symbol table is flagged as follows:

MD	=	Macro Definition
M	=	Label defined in a macro (LOOP and OUT in the example)
V	=	Variable (symbols starting with])
?	=	A symbol that was never referenced

Internally, these are flagged by setting bits 7 to 4 of the symbols length byte:

?=bit 7 MD=bit 5 M=bit 4

Also, bit 6 is set during the alphabetical printout to flag printed symbols, then removed during the numerical order printout. The symbol printout is formatted for an 8Ø column printer, or for one which will send a carriage return after 4Ø columns.

Using MERLIN With 8Ø Column Cartridges

Only the DATA2Ø Videopack 8Ø is supported as of this release. Refer to the VID command in the section on the EDITOR for details on how to use 8Ø column boards with MERLIN.

The Configure ASM Program

Modifyable configuration parameters are given in the DATA line in the BASIC program called "MERLIN". These values are poked into locations $AØØ6-$AØ22. These parameters are described as follows:

LOCATION (HEX)	PRESENT VALUE	PURPOSE AND COMMENTS
AØØ6	Ø	Speed of output to printer. Intended for use with the RS232 port to fine tune out-

	:	: put speed. It can be used to compensate for the lack of printer handshake without having to lower the baud rate. The fastest speed is given by Ø and this should be used for the Commodore printers on the serial port.
A0Ø7	: Ø	: 8Ø column flag for initial entry. Selects DATA2Ø 8Ø-column mode if card is present and this flag is 128 (=$8Ø).
A0Ø8,A0Ø9	: Ø,1Ø ($A0Ø)	: Default source address - must not be below $AØØ.
A0ØA,A0ØB	: Ø,128 ($8ØØØ)	: Default HIMEM and object address.
A0ØC,A0ØD	: Ø,1Ø ($AØØ)	: End of source pointer. Must equal default source address upon entry.
A0ØE	: 94 ($5E)	: Editor's wild card.
A0ØF	: 4	: Number of fields in symbol table printout.
A01Ø	: 47 ($2F)	: Search character "/" for "Update source" question.
A011	: 15 ($F)	: Border color.
A012	: 15 ($F)	: Background color.
A013	: 144 ($9Ø)	: Character color.
A014	: 6	: RS232 control register.
A015	: 16 ($1Ø)	: RS232 control register.
A016,A017	: Ø,Ø	: RS232 optional baud rate (Lo,hi - rarely used)

A018	128 ($80)	Repeat all keys if $80, but not if 0.
A019	60 ($3C)	Number of lines per page (for PRTR).
A01A	7	Number of lines to skip at page end. If this is a 0 a formfeed ($C) is sent to the printer (for PRTR).
A01B	80 ($50)	Number of characters per line (for PRTR).
A01C	128 ($80)	Should be 128 if printer does its own C/R after printing the number of characters in $A01B, else 0.
A01D	0	Sends a linefeed after C/R if this is $D=13 and not if 0. Values other than 13 or 0 must not be used.
A01E	0	If this is 32=$20 then Commodore ASCII will be converted to standard ASCII upon output to the printer. This is used mainly for 3rd party printers on the RS232 port.
A01F	97 ($61)	If this is 97 then characters sent to a printer will be echoed to the screen. If 0 then there is no echo. Other values must not be used.
A020-A022	14,20,31 ($E,$14,$1F)	Default tabs for editor and assembler

Error Messages

BAD OPCODE

Occurs when the opcode is not valid (perhaps misspelled) or the opcode is in the label column.

BAD ADDRESS MODE

The addressing mode is not a valid 651Ø instruction; for example, JSR (LABEL) or LDX (LABEL),Y.

BAD BRANCH

A branch (BEQ, BCC, etc) to an address that is out of range, i.e. further away than ±127 bytes.

NOTE: Most errors will throw off the assembler's address calculations. Bad branch errors should be ignored until previous errors have been dealt with.

BAD OPERAND

An illegally formatted operand. This also occurs if you "EQU" a label to a zero page number after the label has been used. It may also mean that your operand is longer than 64 characters, or that a comment line exceeds 64 characters. This error will abort assembly.

DUPLICATE SYMBOL

On the first pass, the assembler finds two identical labels.

MEMORY FULL

This is usually caused by one of three conditions; Source code too large, symbol table too large, or a PUT file extending beyond HIMEM. See "Special Note" at the end of this section.

UNKNOWN LABEL

Your program refers to a LABEL that does not exist. This also occurs if you try to reference a MACRO definition by anything other than >>>. It can also occur if the referenced label is in an area with conditional assembly OFF. The latter will not happen with a MACRO definition.

NOT MACRO

Reference by >>> to a label that is not a MACRO.

NESTING ERROR

Macros nested more than 15 deep or conditionals nested more than 8 deep.

BAD "PUT"

This is caused by a PUT inside a macro or by a PUT inside another PUT file.

BAD "SAV"

This is caused by a SAV inside a macro.

BAD INPUT

This results from either no input (RETURN alone) or an input exceeding 37 characters in answer to the KBD op-code's request for the value of a label.

BREAK

This message is caused by the ERR opcode when the expression in the operand is found to be nonzero.

BAD LABEL

This is caused by an unlabeled EQU or MAC, a label that is too long or one containing illegal characters.

Special Note - Memory Full Errors

There are four common causes for the "memory full" error message. A more detailed description of this problem and some ways to overcome it follow.

ERROR MESSAGE: "ERR: MEMORY FULL". Generated immediately after you type in one line too many. CAUSE: The source code is too large and has exceeded MERLIN's HIMEM (normally $8000) REMEDY: Raise MERLIN's HIMEM (see the section on the HIMEM command) or break the source file up into smaller sections and bring them in when necessary by using the "PUT" pseudo-op.

ERROR MESSAGE: "MEMORY FULL IN LINE: n". Generated during assembly. CAUSE: Too many symbols have been placed into the symbol table, causing it to exceed availible space. REMEDY: Make the symbol table larger by using the SYM command to lower its beginning address.

ERROR MESSAGE: None, but no object code will be generated (there will be no Object information displayed on the EXEC menu). CAUSE: Object code generated from an assembly would have exceeded the symbol table or BASICs HIMEM. Also can be caused by PUT file being too large. REMEDY: Lower MERLIN's HIMEM or write the object code directly to disk, using the DSK pseudo-op.

When an error occurs on the first pass and while the assembler is processing a PUT file, the error message will indicate the line number preceded by ">" in the PUT file. To find which line of the main program is in effect (which will be the PUT line), simply type "/"<RETURN> and quickly stop the listing. The first line listed will be the active line.

MONITOR

Prompt = $

<-- = Special Commodore Key

EXAMPLE	COMMENTS
$1000: 02 1F 2C -- --	Note that the proper entry format is byte-space-byte etc.
$10001	Disassemble 20 lines beginning at $1000. ASCII indicated at right.
$100011 -- -- --	Disassemble 40 lines beginning at $1000. ASCII indicated at right.
Multiple l's	Continues disassembly at current address.
$1000h	Does a hex dump of 8 bytes at $1000. ASCII indicated at right.
h	h alone continues the dump from current address.

Multiple h's	Dumps multiple 8 byte blocks.
$1000, 1100h	Does a hex dump of the designated range. Note comma is used here.
$1000<--2000,201Fm	Moves range $2000-$201F to $1000. This supports both upward and downward moves.
$1000,2000z	Zeros this range.
$1000<--2000,201Fv	Compares the range $2000-$201F with that starting at $1000 and displays contents of both when differences are found.
$1000g	Jumps to program at $1000. Return by RTS. A BRK will also return with a register display.
$r	Returns to editor.
$q	Returns to EXEC mode. This is a "safe" return even if the zero page locations have been changed.

SOURCEROR

Introduction

SOURCEROR is a sophisticated and easy to use disassembler designed as a subsidiary to create MERLIN source files out of binary programs, usually in a matter of minutes.

Using SOURCEROR

1. SOURCEROR can be LOADed and RUN from BASIC or can be run using MERLIN's "G" command. It requires the standard Commodore 40 column screen to be in use when it is run.

2. SOURCEROR may also be run as follows:

   ```
   LOAD "SOURCEROR.O",8,1
   SYS 12*4096
   ```

   SOURCEROR overwrites part of MERLIN and, after exit, a SYS 52000 will re-enter SOURCEROR not MERLIN, MERLIN must be reloaded.

3. On entry you are asked if you want to load an object file. You must supply the full file name, including the ".O" if any. If this option is selected then the object file will be loaded to a particular location ($A00), and the source address will be placed just following that. MERLIN's menu format will inform you of the range of memory normally occupied by the program. You should start disassembly at this beginning address.

   You will be told that the default address for the source file is $2500. This was selected becuase it does not conflict with the addresses of most binary programs you may wish to disassemble. Just press RETURN to accept this default address. Otherwise, specify (in hex) the address you want.

4. Next, you will be asked to hit RETURN if the program to be disassembled is at its original (running) location, or you must specify in Hex, the present location of the

code to be disassembled. Finally, you will be asked to give the ORIGINAL location of that program.

When disassembling, you must use the ORIGINAL address of the program, not the address where the program currently resides. It will appear that you are disassembling the program at its original location, but actually, SOURCEROR is disassembling the code at its present location and translating the addresses.

5. Lastly, the title page which contains a synopsis of the commands to be used in disassembly will display. You may now start disassembling or using any of the other commands. Your first command must include a Hex address. Thereafter this is optional, as we shall explain.

   At this point, and until the final processing, you may hit RUN/STOP and RESTORE to return to the start of the SOURCEROR program. If you hit RUN/STOP-RESTORE once more, you will exit SOURCEROR and return to BASIC. Using RESET assumes you are using the Autostart monitor rom.

Commands Used in Disassembly

The disassembly commands are very similar to those used by the disassembler in the MERLIN monitor. All commands accept a 4-digit hex address before the command letter. If this number is omitted, then the disassembly continues from its present address. A number must be specified only upon initial entry.

If you specify a number greater than the present address, a new ORG will be created.

More commonly, you will specify an address less than the present default value. In this case, the disassembler checks to see if this address equals the address of one of the previous lines. If so, it simply backs up to that point. If not, then it backs up to the next used address and creates a new ORG. Subsequent source lines are "erased". It is gen-

erally best to avoid new ORGs when possible. If you get a new ORG and don't want it, try backing up a bit more until you no longer get a new ORG upon disassembly.

Command Descriptions

L (List)

This is the main disassembly command. It disassembles 20 lines of code. It may be repeated (e.g. 2000LLL will disassemble 60 lines of code starting at $2000).

If an illegal opcode is encountered, the bell will sound and opcode will be printed as three question marks in flashing format. This is only to call your attention to the situation. In the source code itself, unrecognized opcodes are converted to HEX data, but not displayed on the screen.

H (Hex)

This creates the HEX data opcode. It defaults to one byte of data. If you insert a one byte (one or two digits) hex number after the H, that number of data bytes will be generated.

T (Text)

This attempts to disassemble the data at the current address as an ASCII string. Depending on the form of the data, this will (automatically) be disassembled under the pseudo opcode TXT or DCI. The appropriate delimiter " or ' is automatically chosen. The disassembly will end when the data encountered is inappropriate, when 62 characters have been treated, or when the high bit of the data changes. In the last condition, the TXT opcode is automatically changed to DCI.

Sometimes the change to DCI is inappropriate. This change can be defeated by using TT instead of T in the command.

Occasionally, the disassembled string may not stop at the appropriate place because the following code looks like ASCII data to SOURCEROR. In this event, you may limit the number of characters put into the string by inserting a one or two digit hex number after the T command.

W (Word)

This disassembles the next two bytes at the current location as a DA opcode. Optionally, if the command WW is used, these bytes are disassembled as a DDB opcode. Finally, if W- is used as the command, the two bytes are disassembled in the form DA LABEL-1. The latter is often the appropriate form when the program uses the address by pushing it on the stack. You may detect this while disassembling, or after the program has been disassembled. In the latter case, it may be to your advantage to do the disassembly again with some notes in hand.

Housekeeping Commands

/ (Cancel)

This essentially cancels the last command. More exactly, it re-establishes the last default address (the address used for a command not necessarily attached to an address). This is a useful convenience which allows you to ignore the typing of an address when a backup is desired. As an example, suppose you type T to disassemble some text. You may not know what to expect following the text, so you can just type to L to look at it. Then if the text turns out to be followed by some Hex data (such as $ØD for a carriage return), simply type / to cancel the L and type the appropriate H command.

Q (Quit)

This ends disassembly and goes to the final processing which is automatic. If you type an address before the Q, the address pointer is backed to (but not including) that point before the processing. If, at the end of the disassembly, the disassembled lines include:

```
2341-  4C  Ø3  EØ        JMP  $EØØ3
2344-  A9  BE  94        LDA  $94BE,Y
```

and the last line is just garbage, type 2344Q. This will cancel the last line, but retain the first.

Final Processing

After the Q command, the program does some last minute processing of the assembled code. If you hit RUN/STOP-RESTORE at this time, you will return to BASIC and lose the disassembled code.

The processing may take from a second or two for a short program, to several minutes for a long one. Be patient.

When the processing is done, you are asked if you want to save the source. If so, you will be asked for a file name. SOURCEROR will append the suffix ".S" to this name and save it to disk.

The drive used will be the one used to RUN SOURCEROR. Replace the disk first if you want the source to go on another disk.

To look at the disassembled source, RUN MERLIN.

Dealing with the Finished Source

In most cases, after you have some experience and assuming you used reasonable care, the source will have few, if any, defects.

You may notice that some DA's would have been more appropriate in the DA LABEL-1 or the DDB LABEL formats. In this, and similar cases, it may be best to do the disassembly again with some notes in hand. The disassembly is so quick and painless, that it is often much easier than trying to alter the source appropriately.

The source will have all the exterior or otherwise unrecognized labels at the end in a table of equates. You should look at this table closely. It should not contain any zero page equates except ones resulting from DA's, JMP's or JSR's. This is almost a sure sign of an error in the disassembly (yours, not SOURCEROR's). It may have resulted from an attempt to disassemble a data area as regular code.

NOTE: If you try to assemble the source under these conditions, you will get an error as soon as the equates appear. If, as eventually you should, you move the equates to the start of the program, you will not get an error, but the assembly MAY NOT BE CORRECT. It is important to deal with this situation first as trouble could occur if, for example, the disassembler finds the data AD ØØ 8D. It will disassemble it correctly, as LDA $ØØ8D.

The assembler always assembles this code as a zero page instruction, giving the two bytes A5 8D. Occasionally you will find a program that uses this form for a zero page instruction. In that case, you will have to insert a character after the LDA opcode to have it assemble identically to its original form. Often it was data in the first place rather than code, and must be dealt with to get a correct assembly.

The Memory Full Message

When the source file reaches within $6ØØ of the start of SOURCEROR you will see "MEMORY FULL" and "HIT A KEY" in inverse format. When you hit a key, SOURCEROR will go directly to the final pro-cessing. The reason for the $6ØØ gap is that SOURCEROR needs a certain amount of space for this processing. It is pos-sible (but not likely) that part of SOURCEROR will be overwritten during final processing, but this should not cause problems since the front end of SOURCEROR will not be used again by that point. There is a "secret" override provision at the memory full point. If the key you hit is CTRL-O (for override), then SOURCEROR will return for another command. You can use this to specify the desired ending point. You can also use it to go a little further than SOURCEROR wants you to, and disassemble a few more lines. Obviously, you should not carry this to extremes.

Remember, after exiting to BASIC a SYS 52ØØØ will reenter SOURCEROR (not MERLIN, which has been overwritten).

The LABELER program

One of the nicest features of the SOURCEROR program is the automatic assignment of labels to all recognizable addresses in the binary file being disassembled. Addresses are recog-nized by being found in a table which SOURCEROR references during the disassembly process. This table is on the disk under the name LABELS.O. For example, all JSR $FFD2 instructions within a binary file will be listed by SOURCEROR as JSR CHROUT. This table of address labels may be edited by using the program LABELER.

To use labeler, RUN LABELER.

Labeler Commands

Q:QUIT

When finished with any modifications you wish to make to the label table, press 'Q' to exit the LABELER program. If you wish to save the new file, press 'S'. Otherwise, press ESCAPE to exit without saving the table, for instance, if you had only been reviewing the table.

L:LIST

This allows you to list the current label table. After 'L', press any key to start the listing. Pressing any key will go to the next page; CTRL-C will abort the listing.

D:DELETE LABEL(S)

Use this option to delete any address labels you do not want in the list. After entering the D command, simply enter the NUMBER of the label you want to delete. If you want to delete a range, enter the beginning and ending label numbers, separated by a comma.

A:ADD LABEL

Use this option to add a new label to the list. Simply tell the program the hex address and the name you wish to associate with that address. Press RETURN only, to abort this option at any point.

F:FREE SPACE

This tells you how much free space remains in the table for new label entries.

GLOSSARY

ABORT -terminate an operation prematurely.

ACCESS -locate or retrieve data.

ADDRESS -a memory location.

ALGORITHM -a method of solving a specific problem.

ALLOCATE -set aside or reserve space.

ASCII -industry standard system of 128 computer codes assigned to specified alpha-numeric and special characters. The Commodore uses a somewhat non-standard ASCII. The main difference is the reversal of upper and lower case letters.

BASE -in number systems, the exponent at which the system repeats itself; the number of symbols required by that number system.

BINARY -the base two number system, composed solely of the numbers zero and one.

BIT -one unit of binary data, either a zero or a one.

BRANCH -resume execution at a new location.

BUFFER -large temporary data storage area.

BYTE -Hex representation of eight binary bits.

CARRY -flag in the 651Ø status register.

CHIP -tiny piece of silicon or germanium containing many integrated circuits.

CODE -slang for data or machine language instructions.

CTRL -abbreviation for control or control character.

CURSOR -character, usually a flashing inverse space, which marks the position of the next character to be typed.

DATA -facts or information used by, or in a computer program.

DECREMENT -decrease value in constant steps.

DEFAULT -nominal value or condition assigned to a parameter if not specified by the user.

DELIMIT -separate, as with a: in a BASIC program line.

DISPLACEMENT -constant or variable used to calculate the distance between two memory locations.

EQUATE -establish a variable.

EXPRESSION -actual, implied or symbolic data.

FETCH -retrieve or get.

FIELD -portion of a data input reserved for a specific type of data.

FLAG -register or memory location used for preserving or establishing a status of a given operation of condition.

HEX -the Hexadecimal (BASE 16) number system, composed of the numbers Ø-9 and the letters A-F.

HIGH ORDER -the first, or most significant byte of a two-byte Hex address or value.

HOOK	-vector address to an I/O routine or port.
INCREMENT	-increase value in constant steps.
INITIALIZE	-set all program parameters to zero, normal, or default condition.
I/O	-input/output.
INTERFACE	-method of interconnecting peripheral equipment.
INVERT	-change to the opposite state.
LABEL	-name applied to a variable or address, usually descriptive of its purpose.
LOOKUP	-slang; see table.
LOW-ORDER	-the second, or least significant byte of a two-byte Hex address or value.
LSB	-least significant (bit or byte) one with the least value.
MACRO	-in assemblers, the capability to "call" a code segment by a symbolic name and place it in the object file.
MICROPROCESSOR	-heart of a microcomputer. (In the Commodore the 651Ø chip).
MOD	-algorithm returning the remainder of a division operation.
MODE	-particular sub-type of operation.
MODULE	-portion of a program devoted to a specific function.
MNEMONIC	-symbolic abbreviation using characters helpful in recalling a function.

MSB -most significant (bit or byte), one with the greatest value.

NULL -without value.

OBJECT CODE -ready to run code produced by an assembler program.

OFFSET -value of a displacement.

OPCODE -instruction to be executed by the 651Ø.

OPERAND -data to be operated on by a 651Ø instruction.

PAGE -a 256-byte area of memory named for the first byte of its Hex address.

PARAMETER -constant or value required by a program or operation to function.

PERIPHERAL -external device.

POINTER -memory location containing an address to data elsewhere in memory.

PORT -physical interconnection point to peripheral equipment.

PROMPT -a character asking the user to input data.

PSEUDO -artificial, a substitute for.

RAM -random access memory.

REGISTER -single 651Ø or memory location.

RELATIVE -branch made using an offset or displacement.

ROM -read only memory.

SIGN BIT -bit seven of a byte; negative if value greater than $8Ø.

SOURCE CODE -data entered into an assembler which will produce a machine language program when assembled.

STACK -temporary storage area in RAM used by the 651Ø and assembly language programs.

STRING -a group of ASCII characters usually enclosed by delimiters such as ' or ".

SYMBOL -symbolic or mnemonic label.

SYNTAX -prescribed method of data entry.

TABLE -list of values, words, data referenced by a program.

TOGGLE -switch from one state to the other.

VARIABLE -alpha-numeric expression which may assume or be assigned a number of values.

VECTOR -address to be referenced or branched to.

UTILITIES

Formatter

This program is provided to enhance the use of MERLIN as a general text editor. It will automatically format a file into paragraphs using a specified line length. Paragraphs are separated by empty lines in the original file.

To use FORMATTER, run it from the EXEC mode "G" command. Since it is intended for use with general purpose text files it will overwrite the assembler portion of MERLIN. To assemble a file MERLIN will have to be reloaded. To format a file which is in memory, issue the USER command from the editor.

The formatter program will request a range to format. If you just specify one number, the file will be formatted from that line to the end. Then you will be asked for a line length, which must be less than 25Ø. Finally, you may specify whether you want the file justified on both sides (rather than just on the left).

The first thing done by the program is to check whether or not each line of the file starts with a space. If not, a space is inserted at the start of each line. This is to be used to give a left margin using the editor's TAB command before using the PRINT command to print out the file.

Formatter uses inverse spaces for the fill required by two-sided justification. This is done so that they can be located and removed if you want to reformat the file later. It is important that you do not use the FIX or TEXT commands on a file after it has been formatted (unless another copy has been saved). For files coming from external sources, it is desirable to first use the FIX command on them to make sure they have the form expected by FORMATTER. For the same reason, it is advisable to reformat a file using on<)pp<5ft justification prior to any edit of the file.

Don't forget to use the TABS command before printing out a formatted file.

CHRGEN 8Ø

CHRGEN 8Ø is a 8Ø-column character generator which is designed specifically to allow the use of MERLIN with a 8Ø column by 24 line display on the Hi-Res screen.

TV sets do not provide sufficient resolution for use with CHRGEN 8Ø, thus requiring use of a display monitor for satisfactory results.

To use CHRGEN 8Ø, you must first RUN it from MERLIN's EXEC mode with the "G" command. This will reset the source address to $4ØØØ (above the Hi-Res screen which must be used by CHRGEN 8Ø). This, of course, will delete any source file in memory at the time. Once it has been RUN, you can invoke it at any time by typing "USER" from the editor.

To exit CHRGEN 8Ø, simply use "Q" to return to the EXEC Mode. Upon return to the editor, you can reconnect it by typing "USER" again. To permanently remove CHRGEN 8Ø in order to free up the area normally used by long source listings, you will have to RUN MERLIN again.

If the USER vector has been written over by some other USER routine, it can be reset to point to CHRGEN 8Ø either by RUNing CHRGEN 8Ø again, or by going to the Monitor (use the MON command) and typing in AØØG. The latter assumes, of course, that CHRGEN 8Ø is still intact at $AØØ.

CHRGEN 8Ø includes a version of the FORMATTER program. To implement FORMATTER when CHRGEN 8Ø is connected, just type CTRL-F or F3 from the editor's command mode. NOTE: This command may not be accepted unless something has been listed previously.

CHRGEN 8Ø also includes some keyboard macros. Typing the F5 key followed by certain other keys will produce the keyboard macros. These are presently defined for these keys as:

* > " ' # +

. 2 7 3 X Y D H P @ L S - O C A E

CAUTION: When CHRGEN 8Ø is up, you must not load any source file longer than 96 sectors or it will overwrite MERLIN and bomb the system. Text files do not present this danger since they are never allowed to go beyond HIMEM.

XREF, XREF.XL and STRIP

Utility programs XREF, XREF.XL and STRIP provide a convenient means of generating a cross-reference listing of all labels used within a MERLIN assembly language (i.e., source) program.

Such a listing can help you quickly find, identify and trace values throughout a program. This becomes especially important when attempting to understand, debug or fine tune portions of code within a large program.

The MERLIN assembler by itself provides a printout of its symbol table only at the end of a successful assembly (provided that you have not defeated this feature with the LST OFF pseudo op code). While the symbol table allows you to see what the actual value or address of a label is, it does not allow you to follow the use of the label through the program.

This is where XREF, XREF.XL and STRIP come in.

XREF gives you a complete alphabetical and numerical printout of label usage within an assembly language program with a length of up to approximately 1,ØØØ lines (heavily commented) or 2,ØØØ lines (lightly commented).

XREF.XL handles "extra-large" files of up to three or four times the size of those handled by XREF by storing the generated cross reference table on disk and printing it out later.

STRIP provides a method of reducing file size by removing comments from source code.

Sample MERLIN Symbol Table Printout:

Symbol table - alphabetical order:

```
   ADD     =$F786      BC      =$F7BØ      BK      =$F7Ø6
```

Symbol table - numerical order:

```
   BK      =$F7Ø6      ADD     =$F786      BC      =$F7BØ
```

Sample MERLIN XREF Printout:

Cross referenced symbol table - alphabetical order:

```
     ADD     =$F786  1Ø1   185*
     BC      =$F7BØ  9Ø    2Ø7*
     BK      =$F7Ø6  1Ø4   121*
```

Cross referenced symbol table - numerical order:

```
     BK      =$F7Ø6  1Ø4   121*
     ADD     =$F786  1Ø1   185*
     BC      =$F7BØ  9Ø    2Ø7*
```

As you can see from the above example the "definition" or actual value of the label is indicated by the "=" sign, and the line number of each line in the source file that the label appears in is listed to the right of the definition. In addition, the line number where the label is either defined or used as a major entry point is suffixed ("flagged") with a "*".

An added feature is a special notation for additional source files that are brought in during assembly with the PUT pseudo opcode: "134.82", for example, indicates line number 134 of the main source file (which will be the line containing the PUT opcode) and line number 82 of the PUT file, where the label is actually used.

XREF Instructions

1. Get into MERLIN's Executive Mode, make sure you've S)aved the file that you're working on and select the D)rive no. that the MERLIN disk is in.

2. Type "G" and then "XREF" <RETURN> (Your file in memory will now be erased.)

3. Re-L)oad your file. Initialize your printer with the appropriate PORT or PRTR command (XREF is usually, but not necessarily, a printer oriented command). Use PRTR Ø if you just want screen output.

4. Type in the appropriate USER command:

   USER Ø -Print assembly listing and alphabetical cross reference only. (USER has the same effect as USER Ø).

   USER 1 -Print assembly listing and both alphabetical and numerically sorted cross reference listings.

   USER 2 -Do not print assembly listing but print alphabetical cross reference only.

   USER 3 -Do not print assembly listing but print both alphabetical and numerical cross reference listings.

USER commands Ø-3 (above) cause labels within conditional assembly areas with the DO condition OFF to be ignored and not printed in the cross reference table.

There are additional USER commands (4-7) that function the same as USER Ø-3, except that they cause labels within conditional assembly areas to be printed no matter what the state of the DO setting is. The only exception to this is that labels defined in such areas and not elsewhere will be ignored.

NOTE: You may change the USER command as many times as you wish (e.g., from USER 1 to USER 2). The change is not per-

manent until you enter the ASM command (below).

5. Enter the ASM command to begin the assembly and printing process.

CAUTIONS for the use of XREF

XREF works by examining the listing output of the assembler. On the second assembly pass, it builds a cross reference list beginning at HIMEM instead of creating object code there. (If direct assembly to disk is selected by the DSK opcode, however, the object code will be generated). The list uses six bytes per symbol reference which can use up available memory very quickly. Thus, on long files, you should set HIMEM as low as possible. (The WØ command can be used to find the end of the source file, which represents the lowest position you can set HIMEM).

Since the program requires assembler output, code in areas with LST OFF will not be processed and labels in those areas will not appear in the table. In particular, it is essential to the proper working of XREF that the LST condition be ON at the end of assembly (since the program also intercepts the regular symbol table output). For the same reason, the F3 flush command must not be used during assembly. The program attempts to determine when the assembler is sending it an error message on the first pass and it aborts assembly in this case, but this is not 1ØØ% reliable.

Macros require special consideration. Since the syntax in these structures can become very complicated, XREF may get confused and cause assembly to stop. This usually happens when lines containing the >>> pseudo opcode are followed by string literals or parentheses and you have chosen to suppress printing the expanded form of the macro in your assembled listing with the EXP OFF pseudo opcode. You can get around this problem by printing out the assembly listing first in the usual manner (with the symbol table suppressed by the LST OFF pseudo opcode) and then printing out just the cross reference table with EXP ON and using USER 2 or 3.

Another thing to look out for when using macros is the fact

that labels defined within macro definitions have no global meaning and are therefore not cross-referenced.

```
DEF     MAC                 <---Macro definition
        CMP  #]1
        BNE  DONE
        ASL
DONE    <<<
------------------------    <---Beg. of program
        >>>  DEF.GLOBAL     <---Macro call
```

In the above example, variable GLOBAL will be cross referenced, but local label DONE will not.

XREF.XL Instructions

XREF.XL is designed to handle files three to four times as large as those handled by XREF.

To use XREF.XL, just follow the same five steps in the XREF instructions explained previously, substituting "XREF.XL" for "XREF" in step 2.

XREF.XL works in a manner similar to XREF, except that it writes the cross reference label table to disk in a file called X.R.FILE (You can delete this file when you are done with the table). At the end of assembly, this file is loaded from disk and placed in memory, overwriting your source file. As explained in step 1, make sure that you've saved your source file first, because the source file will be deleted from memory when you return to the editor.

CAUTIONS for the use of XREF.XL

-The source file will be deleted from memory as explained above when you return to the editor. Make sure that you have saved your file first.

-Consider using a blank disk when using XREF.XL. The disk file generated, X.R.FILE, can become quite large.

-The cross reference label table X.R.FILE is written on the disk in the disk drive last used. If your source file contains PUT directives, you will have to make sure XREF.XL can find the additional source files by either moving the files onto the blank disk or by specifying a drive parameter in the PUT directive.

-Certain things can cause the XREF program (or CHRGEN 8Ø) to be "disconnected". They can be reinitialized by going to the monitor and entering AØØG.
-Unlike XREF, the setting of HIMEM does not affect XREF.XL. While building the cross reference table, XREF.XL checks to see if it will fit in the space from the source address (approximately) to the SYM address, if specified, or to $AØØØ if not. If it is too large, XREF.XL will quit with an OUT OF MEMORY message.

-XREF.XL will quit with an ILLEGAL DSK ATTEMPTED error message if it finds a DSK pseudo op code in your source file. A handy way of avoiding this problem while at the same time maintaining the same line numbers in the source file is to use the editor to change any DSK directives into comments.

XREF A and XREF A.XL

These are ADDRESS cross reference programs and are handy when you have lots of PUT files. Since these need only four bytes per cross reference instead of six, they can handle considerably larger sources. Also the "where defined" reference is not given here because it would equal the value of the label except for EQUated labels where it would just indicate the address counter when the equate is done. This also saves

considerable space in the table for a larger source.

STRIP

Very long source files, or ones that contain numerous comments, may require too much memory for the cross reference table to be generated. In this case, assembly will stop with the OUT OF MEMORY error message.

The utility program STRIP allows you to cross reference files approximately twice as large by removing comments from the source file.

To use STRIP, follow the following procedure:

1. Make sure that you have a copy of your commented source file in memory and that you have S)aved a copy of it on disk.

2. Enter the E)ditor, put a LST OFF at the end of your source file and ASM it.

3. Remove the LST OFF statement at the end of your program (important!).

4. Q)uit the editor, select the D)rive with MERLIN in it and type "G" then STRIP <RETURN>.

5. You may now use the XREF and XREF.XL procedures as outlined above.

CYCLE TIMER

This utility causes a machine cycle count to be displayed during assembly. To use it you must run CYCLE TIMER using the EXEC mode "G" command. This must be done PRIOR to loading the source file, since it resets the file pointers. You must type the desired PRTR command, then USER and then ASM. The cycle times will be printed to the right of the comment field and will look like this:

5,Ø326 or 5'Ø326 or 5",Ø326

The first number displayed is the cycle count for the current instruction and the last is the accumulated total (in decimal). The single quote indicates a possible added cycle, depending on certain conditions. If this appears on a branch instruction then it indicates that one cycle should be added if the branch occurs. For non-branch instructions, the single quote indicates that one cycle should be added if a page boundary is crossed. A double quote occurs only on branch instructions and indicates that two cycles should be added if the the branch is taken. (The CYCLE TIMER program has determined that the branch would cross a page boundary in this event.)

There are four locations, $AØ3-$AØ6, that can be adjusted by the user. You may adjust the position at which the count will be printed, the number (2, 4, or 6) of digits in the accumulated total, the averaging default, and the USR enable flag.

The byte at $AØ3 contains a number 2 less than the column in which the cycle count will start printing. The count, however, is always printed after the comment. To have the count printed as far to the right as possible this byte should be set to the total number of columns minus the number (2,4 or 6) of digits in the accumulated total less 5. This is presently set to $47 which is appropriately for a 8Ø column printer.

The byte at $AØ4 should contain $8Ø, $4Ø or Ø depending on whether you want 2, 4, or 6 digits in the accumulated total count.

The byte at $AØ5 should contain either Ø or $8Ø depending on whether you want the indeterminable added cycles averaged in the total count or not.

The byte at $AØ6 should be either Ø or $8Ø depending on whether you want the internal USR routine to be enabled or not. If the USR routine is enabled then the USR opcode will reset the accumulated total to zero. If $AØ6 contains $8Ø then the USR opcode is handled just as if CYCLE TIMER is not

being used.

If your printer allows it, we suggest using CYCLE TIMER with the printer set at 96 characters per line rather than the usual 8Ø. This will allow assembly of files ordinarily formatted for 8Ø columns without having the timer data overflow the lines. Of course, the byte at $AØ3 must be set accordingly. You must also set the number of columns for the PRTR command to 96 (location $AØ1B).

CONFIL

This program can be used to do the major portion of conversion of a source file from the CBM ASSEMBLER64 to MERLIN format. To use it, read the file into MERLIN with the 'R' EXEC command. Then use the EXEC 'G' command to run CONFIL. The main conversion done by the program is to change the "character set 1" (UC/graphics) used by the CBM ASSEMBLER64 (which becomes lower case in MERLIN) to the "character set 2" (UC/LC) used by MERLIN. Lower case characters are converted to UC until a comment or literal (single quotes) is met. Note that although MERLIN would accept lower case for opcodes and labels, it requires upper case for hex digits A-F. Literals (eg. LDA #'a') which are in UC in the CBM ASSEMBLER64 are properly lower case in MERLIN to assemble the same code so they are not converted.

The pseudo opcodes .BYTE, .DBYTE, .PAGE, .SKIP, .END, .MAC, DFB, DDB, PAG, SKP, END, MAC, <<<, and DA. Other pseudo opcodes and the "* =" lines will have to be hand converted since their syntax is substantially different from that used by MERLIN. Also the macro name in a .MAC definition will have to be moved to the LABEL column.

CONFIL does change the macro variables ?1-?8 to]1-]8 but the user must change macros and macro calls to reflect the fact that MERLIN uses ordinary labels to label lines in a macro definition (they are local) and the commas used to seperate label assignments in macro calls will have to be changed to the semicolons used by MERLIN.

RESET

This resets source pointers to the usual value of $A00 and moves any source in memory down to that point. This utility is intended for use after one of the XREF programs or CHRGEN 80, if desired. To use it just run it via the EXEC mode "G" command.

MISCELLANEOUS ROUTINES

A number of subroutines are on the disk as SEQ files. These are often used routines and are intended as PUT files or to be appended to programs and modified as desired. Most of them require certain equates to have been defined, or local variables declared in a VAR statement. Read them for details on their usage.

MERLIN AND THE C64 LINK

MERLIN contains a version called MERLIN.L which is compatible with the Richvale Telecommunications C64 LINK. To use it you must run the RELOCATOR 5.5 program on the C64 LINK disk and select "PET EMULATOR" mode. The Pet emulator itself should NOT be run. Then type DLOAD "MERLIN.L" and RUN.

If the C64 LINK is used with a disk drive on the SERIAL port there is total compatibility except for the lesser memory and the DATA 20 80-column card, which is not compatible with the C64 LINK.

If the C64 LINK is used with NON-SERIAL disk drives then some MERLIN functions such as the ERR (in the editor), CATALOG, and disk error reporting in general will not be available. To do a catalog you must exit to BASIC and issue the C64 LINK CATALOG command, then return to MERLIN with a SYS 52000.

The USR opcode cannot be used with the C64 LINK due to memory conflicts.

SOURCEROR will work IF it is run using MERLIN's RUN command, but will be considerably limited in memory available for its use. Also, there is some possibility of overwrite of the C64 LINK code, in which case the system will crash. Therefore, it is NOT advisable to force continuation of disassembly with the Control-O override.