SYMBOL MASTER™

Multi-Pass Symbolic Disassembler

for

Commodore 64

and

Commodore 128
(128 mode)

Serial No. 621

Schnedler Systems
1501 N. Ivanhoe St.
Arlington, VA 22205
(703) 237-4796

COPYRIGHT

The Symbol Master program and this manual are both copyrighted under United States and International copyright laws. All rights are reserved.

COPY PROTECTION

Despite dire warnings from others, Symbol Master is not, and has never been, copy protected. There are two main reasons: (1) I doubt there is a copy protection scheme which cannot be broken in any event, especially with a tool as powerful as Symbol Master. (2) Particularly with a programmer's utility, copy protection often gets in the way of ease of use. I have assigned a higher priority to making Symbol Master easy to use.

Since the disk is not copy protected, you are encouraged to back it up for your own protection in the event the disk becomes damaged. This does not authorize you to make copies for any other purpose, as such action would clearly be contrary to the copyright.

Finally, I like to believe that purchasers of this product will respect the effort which went into it, and not cheapen it by giving or trading it away. The original version 1.0 took five months to write, and the upgrade to version 2.0 took another five months. This involved extremely long periods of intense effort. Would-be pirates should be aware, however, that I am a full time practicing attorney specializing in Patent, Trademark, Copyright, Trade Secret, and Unfair Competition law, and related matters. I am a partner in the law firm of Kerkam, Stowell, Kondracki & Clarke, 1235 Jefferson Davis Highway, Suite 411, Arlington, Virginia 22202.

DISCLAIMER

Steven C. Schnedler and Schnedler Systems make no warranties, express or implied, as to the quality, performance, merchantability or fitness for any particular purpose of the Symbol Master software and this manual. All risk relating to its performance, reliability, and suitability for a given purpose is with the buyer. We will not be held liable or responsible for direct, indirect, incidental, or consequential damages resulting from any defect in the software or manual under any condition.

SUPPORT

The above, of course, is for our own protection. Nevertheless, we believe Symbol Master is a superior product, and one you will find a valuable tool. We have fully supported this program since its inception, and in the past we have provided "bug fixes" to every customer at no charge. Please let us know if you do have problems. We will do everything we can.

TRANSFER

Your name is on the disk. If we spelled your name wrong, or if you wish to transfer ownership to another person, send us back your original disk, with instructions, and we will re-groove it at no charge.

Schnedler Systems
1501 N. Ivanhoe St.
Arlington, Virginia 22205

(703) 237-4796

SYMBOL MASTER

Multi-Pass Symbolic Disassembler

for

Commodore 64

and

Commodore 128
("Native" 128/8502 mode)

This edition of the Manual
Revised February 2, 1986
for
Symbol Master Version 2.0

Dedications:

To My Wife,
Without whose urgings Version 2.0
might never have been completed

and

To the PTD-6510 Symbolic Debugger,
The most amazing tool I have ever
used and without the power of which
neither Symbol Master 1.0 nor 2.0 might
not have been possible

Program and Manual by:

Steven C. Schnedler

SCHNEDLER SYSTEMS
1501 N. Ivanhoe St.
Arlington, Virginia 22205

(703) 237-4796

TABLE OF CONTENTS

Chapter 1: QUICK START PROCEDURE 4-6

1.Ø Preliminaries 4
1.1 C64 Quick Start 4
1.2 C128 Quick Start 6

Chapter 2: SYMBOL MASTER OVERVIEW 7-1Ø

2.Ø Description of a Symbolic Disassembler .. 7
2.1 Symbol Master Disk Output 7
2.2 Summary of Use 8
2.3 Other Features 1Ø

Chapter 3: ASSEMBLERS SUPPORTED 11-12

3.Ø Introductory Comments 11
3.1 The Assemblers More Specifically 11
3.2 Relevant Consumer Information 12

Chapter 4: EQUIPMENT CONFIGURATION 13

Chapter 5: SYMBOL MASTER MEMORY USAGE -- C64 Version ... 14-16

5.Ø In general 14
5.1 Zero-page Memory Usage 15
5.2 Other Memory Areas Used 15
5.3 Symbol Master Environment 15
5.4 Co-residency with other programs 16

Chapter 6: SYMBOL MASTER MEMORY USAGE -- C128 Version 17

6.Ø In General 17
6.1 RAM Bank Ø Version 17
6.2 RAM Bank 1 Version 17

Chapter 7: INTRODUCTION TO THE SYMBOL MASTER
COMMAND SCREEN EDITOR 18-19

7.1 Overall Guidelines 18
7.2 Editor Screen Line Summary 19
7.3 Scope of Command File 19

Chapter 8: SCREEN EDITOR LINE 1 -- COMMAND INPUT LINE ...2Ø-25

(Seventeen commands listed and described)

Chapter 9: SCREEN EDITOR LINE 3
MISCELLANEOUS PARAMETER LINE 26

Chapter 1Ø: SCREEN EDITOR LINE 4
CONTROL AND EQUATE FILENAMES LINE 27

Chapter 11: SCREEN EDITOR LINES 6-25
MEMORY BLOCK DEFINITION LINES ..... pages 28-31

11.Ø Formal Syntax Definition 28
11.1 Effect of block filenames 29
11.2 Relationship of Block Memory Ranges 3Ø
11.3 The "Fix" Commands 3Ø
11.4 Tip on entering lines 3Ø
11.5 Inserting and deleting 31
11.6 ASCII Interpretation 31

Chapter 12: DOS COMMANDS 32

Chapter 13: SYMBOL MASTER FILENAMES 33

Chapter 14: ARBITRARILY GENERATED LABELS 34-35

Chapter 15: THE LABEL NAME EDITOR 36-41

15.Ø General Concepts and Purpose 36
15.1 Single- and Two-Byte Named Labels 36
15.2 Brief Introduction to the Editor 37
15.3 Syntax of Label Name Entries 37
15.4 Entering, Deleting, Viewing 37
15.5 Label Editor Commands 38
15.6 Provided Label Name Files 4Ø
15.7 Miscellaneous Notes 41

Chapter 16: "BIT-SKIP" HANDLING 42-43

Chapter 17: UNDOCUMENTED AND 65CØ2 OP-CODES 44-45

17.1 The 65Ø2 Undocumented Op-Codes 44
17.2 Modifying the Disassembler 45
17.3 The 65CØ2 Documented Op-Codes 45

Chapter 18: PRELIMINARY INVESTIGATION
THE MODIFIED MACHINE-LANGUAGE MONITOR 46-49

18.1 The Machine-Language Monitor 46
18.2 The Added Commands 47
18.3 Other Relevant Commands 48
18.4 Preliminary Investigation 48
18.5 Remaining Monitor Commands 49

Chapter 19: AUTOBOOT PROGRAMS AND COPY PROTECTION 5Ø-53

19.Ø C-64 Autoboot Programs 5Ø
19.1 Caution on Load Addresses 51
19.2 Copy Protected Programs 51
19.3 Mangled Disk File Names 52
19.4 File Reading Program 52
19.5 C-128 Boot Programs 53

Chapter 2Ø: SPECIFIC ASSEMBLER NOTES 54-57

2Ø.1 MAE Assembler 54
2Ø.2 PAL64 Assembler 55
2Ø.3 Develop-64 55
2Ø.4 Commodore Assembler 56
2Ø.5 LADS Assembler 56
2Ø.6 Merlin Assembler 57
2Ø.7 Panther Assembler 57

Chapter 21: PREPARED EXAMPLES 58-6Ø

Chapter 22: MISCELLANEOUS NOTES 61-63

22.Ø Further notes on disk 61
22.1 C128 Capabilities 61
22.2 C128 8Ø-Column "Fast" Mode 61
22.3 Error conditions 61
22.4 Defining Blocks 62
22.5 Printing of Messages 62
22.6 Halting and Pausing Execution 63
22.7 Random Tips 63
22.8 Disassembling C64 Cardridge ROMs 63

1.Ø Preliminaries

This chapter is for anyone who wants to start and do something first, and read the explanation and documentation later. However, even before doing the following procedures, we suggest that you immediately make a working copy of your Symbol Master disk, and put the original safely away. Most everyone has a disk back-up program by now. In case you don't, you will find a slow, but reliable, public domain program entitled "BACKUP 1541.BAS" on your Symbol Master disk.

The examples below take you through a disassembly of Commodore's "DOS 5.1", which comes on the "Test/Demo" disk packed with 1541 disk drives. If you have a disk drive, you undoubtably already have "DOS 5.1". To be sure, a copy is included on the Symbol Master disk. Even if you have one, we suggest that you use the one on the Symbol Master disk, in case there are different versions. The first example is for the C-64 (or a C-128 in C-64 mode). The second example is for the C-128 in 128 mode.

There are several other more advanced examples later in this documentation booklet, including disassembly of the Basic ROM, the Kernal ROM, the 1541 disk drive ROM, and disassembly of Symbol Master itself!

Two purposes are served by these and the later examples. First, they do serve the "quick start" purpose mentioned above. Second, the examples are selected with a view towards illustrating various techniques. Symbol Master is a powerful tool. A corollary to that power is that it accepts many commands in support of the various options. Initially it can seem quite complex, but you will quickly become comfortable with it. You are urged to consider the purpose and effect of every single line which appears on your editor screen relating to these examples.

1.1 Commodore 64 Quick Start -- Example 1

(1) LOAD and RUN the Basic boot program "BOOTSM64.BAS":

```
LOAD "Ø:BOOTSM64.BAS",8
```

then

```
RUN
```

(2) Select menu option #3, Symbol Master at $9ØØØ. The Boot program will then load "SM64EDIT$9.EXE", and then SYS to its cold start address $9ØØØ (9*4Ø96 = 36814 decimal). "SM64EDIT$9.EXE" will in turn automatically load two main disassembler modules, "SM64$D.EXE" and "SM64$Λ.EXE" under the I/O area and under Kernal and Basic ROMs.

Note: If in step (1) above you modify the Boot program to load "SM64EDIT$x.EXE" from a disk device number other than 8, specifically 9, 1Ø or 11, then the "SM64EDIT$n.EXE" module will automatically use that device number to load the two main modules.

(3) When loading is completed and you are presented with the "Cold Start" message, press any key for the command editor screen.

(4) With the cursor anywhere on the top screen line, preferably the upper left corner, type the following and enter using the RETURN key:

```
L "Ø:DOS 5.1"
```

This will do a normal Kernal LOAD of the DOS 5.1 program file (the target code) from disk to memory starting at its normal load address $CCØØ determined by the header consisting of the first two bytes of the file. The ending address (plus one) will be reported to you at the conclusion of the load, and the cursor will be left on the following blank line. Any load error will be reported as a Kernal I/O error number.

(5) With the cursor still on a blank line press the RETURN key to re-initialize the screen editor, and clear the top line for another command.

(6) Again on the top line, enter the following; note quotes are not used this time:

```
G Ø:DOS5.1.CMD
```

This will Get an appropriate prepared "command" file from disk with the same effect on the editor screen as though you had typed it in. The screen will now have the precise instructions needed to properly disassemble "DOS 5.1". (A "command" file contains the parameters passed from the editor to the disassembler itself telling the disassembler, among other things, where in memory to find the code to be disassembled.)

(7) Clear the top line using either of these methods: (a) Spacebar over all text and then run the cursor back; or (b) SHIFT/CLR to clear the entire screen, and then RETURN to re-initialize the Symbol Master editor.

(8) On the top line enter the following:

```
U Ø:DEFAULT64.LBL
```

This will load a prepared label name file into the disassembler including label names to be used during the dissassembly. (Using the separate Label Name Editor, described in a later chapter, you can create and modify label name files appropriate to particular disassemblies which you are doing.)

(9) The default assembler is MAE. If you use a different assembler, change to that one now so that the pseudo-ops generated on the listing will be ones familiar to you: Run the cursor to the third screen line, the one which presently reads "nn S A MAE", over the "M" in "MAE" and type the three letters which name your assembler. With Symbol Master Version 2.Ø, your other choices are PAL for PAL, D64 for Develop-64, CBM for the Commodore Macro Assembler, LAD for LADS, MER for Merlin, or PAN for Panther. Enter the assembler name using the RETURN key.

(1Ø) Run the cursor back to the top screen line, clear the line using one of the methods in paragraph (7) above, and then Run the disassembler by entering:

```
R
```

This command means Run, with default options. You will see the disassembly output to the screen, followed by a cross referenced label table.

(11) For a printer output of the same thing, enter on the top screen line:

```
R PCX
```

The options following the Run command respectively mean output to Printer, output disassembly of Code (including data tables), and generate a cross-referenced (Xr) symbol table.

(12) If you prefer the printer output to skip over perforations, position your printer page at the top of a sheet, and enter the following:

```
R PCXF
```

The additional option means to Format by inserting six blank lines after every sixty printed lines.

(13) If you would like to generate a source code file to disk, insert a formatted disk into your drive, and enter the following command on the top line:

```
R D
```

The "D" option means output to disk. Your assembler will be able to later read in and assemble the generated file, assuming you selected the appropriate assembler option in step (7) above.

1.2 Commodore 128 Quick Start -- Example 2

The same example in C-128 mode is very similar. The instructions here are briefer, and assume you have read the above.)

(1) For this example, put the C-128 in the 40-column mode. (80-column "fast" mode procedures are described in Chapter 22.

(2) LOAD then RUN the program file "SM128/BANK0", which for convenience is set up like a basic program file:

```
DLOAD "SM128/BANK0"
```

(3) After you enter RUN you will be presented with the "Cold Start" message. Press any key for the command editor screen.

(4) With the cursor anywhere on the top screen line, preferably the upper left corner, enter:

```
L1 "0:DOS 5.1"
```

The only difference between this and the corresponding step (4) of the C-64 example is the "1" after the "L". The effect of the "1" is to do a Load into C-128 RAM Bank 1, necessary to avoid conflict since Symbol Master is in RAM Bank 0.

(5) Refer to the C-64 example above, and do steps (5), (6), (7), (8) and (9) exactly the same way. The particular RAM bank makes no difference.

(6) This should already be set for you, but refer again to the third screen line. The second field of the prepared example command file should be "1", which tells Symbol Master which of the sixteen logical bank configurations to disassemble from. "0" through "F" are acceptable entries. (The usual default is "F".) The entry "1" is consistent with the target code loaded into RAM bank 1.

(7) Referring again to the C-64 example, Run the disassembler using steps (10), (11), (12), (13) and (14) in exactly the same way.

2.Ø Description of a Symbolic Disassembler

Nearly every programmer has a machine-language monitor with simple disassembly capability, such as the modified version of MICROMON included on the Symbol Master disk. Such programs allow you to scroll forward and backward through memory while viewing the op-codes and operands in hex, the 65Ø2 instruction set mnemonic, and the operand in hex. Lacking, however, are labels on those program lines which are referenced by other lines in the program. Machine-language monitors also correspondingly include a simple assembler. A simple assembler is satisfactory for very short programs and for program patches during debugging, but for any significant programming work a symbolic assembler is greatly preferred.

In general, a symbolic disassembler compares to a simple disassembler in the same way that a symbolic assembler compares to a simple assembler in a machine-language monitor.

A symbolic disassembler assigns lables to those lines which are referenced by other lines, and uses those same lables in the referencing lines.

2.1 Symbol Master Disk Output

A significant feature of Symbol Master is its ability to rapidly generate source code files to disk which can in turn then be read by your assembler. Using your assembler's editor, generated labels can be changed to more meaningful names as you begin to understand the code. (You can also use the label name editor to minimize this step.) Most assembler editors have search and replace capability which allows you to automatically go through a source code file and reliably change all occurrances of a given label to a name of your choice. Comments can be added. Instructions can be inserted and deleted, and the program generally modified. Programs can be re-assembled to a different origin address. Portions can be incorporated into your own programs.

Symbol Master optionally generates source code files directly compatible with each of the assemblers supported. This means that the files are of the right type, and the appropriate pseudo-ops are used. The result is indistinguishable from a source code file generated by the assembler's own editor itself. Except in rare cases, Symbol Master source code file output can be immediately re-assembled using your assembler to exactly recreate the object code Symbol Master started with.

Another disk output option, available with each assembler option, is to generate a single file (S option) or multiple files (M option). The Single file option is used for relatively shorter programs, and results in a single source code file which includes the equates, the code, and any data tables. The Multiple file option is used for relatively longer programs, and avoids generating source code files which are too long for your assembler to handle in memory all at once. Even if your assembler can handle long files, for editing purposes it is usually less cumbersome to break a the source code of a program into smaller file modules.

With the Multiple file option, a Control file and an Equate file are generated, and at least one Module file. You select where one module is to end, and the next to begin. There are really no constraints in this regard insofar as Symbol Master is concerned. Significantly, you can switch between blocks of code and blocks of data tables without causing a new file to be generated, unless you wish. You indicate the beginning of a new file by assigning a file name on the block line.

We recommend that you limit each of your files to 1K of object code, or less. Counting in hex, it is often difficult to keep track of how much 1K is. For example, from $1000 to $1FFF is 4K. Presented below are the boundaries of the four 1K portions of this 4K:

$1000 through $13FF
$1400 through $17FF
$1800 through $1BFF
$1C00 through $1FFF

2.2 Summary of Use

To use Symbol Master in the C-64, the editor version you select is loaded into a 4K block of memory and cold started. For convenience, four versions are included, identical except for where they load: $1000 through $1FFF, $8000 through $8FFF, $9000 through $9FFF, and $C000 through $CFFF. The editor in turn loads the main disassembler portions to RAM under the Basic and Kernal ROMs, and under the I/O area.

There are two versions for the C-128. The RAM Bank 0 version loads beginning at $1C01, the start of the text area for Basic programs. The cold start address is $1C12 (decimal 7186). The RAM Bank 1 version loads beginning at $0400, and the cold start address is also $0400 (decimal 1024). Either C-128 version uses all of the RAM in the particular bank in which it is loaded (although the higher RAM will remain untouched unless you are doing an extremely long disassembly which generates a large label table), and can disassemble from any of the 16 logical bank configurations, including its own.

From the Symbol Master command screen, the target program is loaded into RAM for study. The target program can be loaded either to where it normally loads as determined by its own two-byte header, or it can be loaded to any specified starting address. Symbol Master can correctly disassemble programs, even if they are loaded other than where they are intended to run, by automatically applying an offset correction. This is a particularly useful capability when studying automatic "Boot" programs which in the C-64 are normally loaded from Basic via

LOAD "<filename>",8,1

and which then proceed to literally take over control of your computer.

Thus Symbol Master and the target program for study are co-resident in memory. In addition, it is useful to also have a machine-language monitor program in memory, such as the included specially-modified version of MICROMON (C-64 only).

Symbol Master does not disassemble directly from disk files, and there are good reasons for this. A significant reason is speed. In order to do everything it does, Symbol Master always makes at least four passes through the code being disassembled, five passes for MAE. To read through a disk file this many times could be quite time consuming. Moreover, when generating the optional cross-referenced label listing, Symbol

Master passes through the entire target code once for each label. Another reason is that the "header" written as the first two bytes of a program file is usually, but not always, the load address. Program files can be loaded via the Kernal LOAD routine to any address when a relocated load is selected. Copy protection schemes often do this. Yet another reason is that Symbol Master's approach allows you to correctly disassemble programs which have data tables embedded between blocks of code. A further reason is that Symbol Master can disassemble together blocks of code which run together, but which are not contiguous in memory.

If you load the Symbol Master editor in the C-64 at $C000 through $CFFF, this leaves $0800 through $9FFF free for your target programs for study. This is 38K. In most cases you will also want to load MICROMON at either $1000 through $1FFF or $9000 through $9FFF, which will still leave at least 32K for your target programs under study. In the C-128, Symbol Master will reside in either RAM bank, leaving the entire other RAM bank for target code. Again, it is not necessary for a target program to be loaded where it normally runs. Symbol Master has the capability of automatically applying an offset to make the necessary adjustment as it examines each instruction.

At some point, the simple disassembler in MICROMON (or the C-128 MONITOR) is used to locate those parts of the program which do not disassemble correctly, i.e., are data tables. The starting and ending address of each block of code and data tables are noted. Care should be taken at transistion points, particularly when switching from data tables to code, to determine the precise address where code begins. The particular starting address selected can make the difference between a correct disassembly from the beginning, and junk bytes to start.

Symbol Master does not mind going through data tables after being told they are code. However, the results will not be meaningful to you, and the generated list of equates to external label references will be cluttered with spurious labels.

When looking at the data tables with MICROMON, try to determine whether they are ASCII data or not, and whether they are two-byte Word data or three-byte Table data. Each of these is an option to specify to Symbol Master. Also, Symbol Master should definitely be sent through data tables, because Symbol Master will assign labels to those bytes which are referenced by instructions in the blocks of code. The bytes themselves will be left in hex byte form, with ASCII interpretation in the comment field if this option is selected. Moreover .WORD tables (two-byte address constants) will cause Symbol Master to generate a proper label applicable to each byte referenced by a .WORD entry, often a routine entry.

When this is completed, the starting and ending addresses of each block are given to Symbol Master using the screen editor portion below the dash line. Each line is for one block. Twenty lines fit on the screen at once. If more than twenty are used, the window will scroll up and down via the cursor up/down key, up to the maximum of 99 blocks.

All of this information entered to the screen editor can be saved to disk via the "P" command, for Put. It can be retrieved for later modification via the "G" command, for Get. Thus, if you are working out a complex disassembly, your efforts can be saved between sessions. Also, you can dump the entire editor screen to your printer via the "H" command, for Hardcopy.

There are several other optional choices to make, discussed below in the Screen Editor Section, and you are then ready for disassembly to the output device of your choice: Screen, printer or disk. The cross-referenced label table can be output to the Screen or Printer.

It is always best to direct the output first to the screen, to see if any commands need to be changed. The display can be slowed down with the CTRL key, or paused with the STOP key and restarted with RETURN. In the C-128, the "NO SCROLL" key also works. Or the screen output display can be paused with the STOP key, and then aborted with the DEL key, which returns via warm start to the editor. You will find Symbol Master to be quite fast, and this facilitates iteration to find the best instructions to issue to the disassembler for the most meaningful disassembly.

CAUTION. With printer and disk output there is no pause and restart option. The STOP key causes an abort. You may have to press the STOP key several times because it is not scanned very often while the serial bus is active, particularly to the printer.

2.3 Other Features

Symbol Master has many convenience features which you will grow to appreciate. Some of these are briefly mentioned below, and described in detail later:

(1) A scrolling, full-screen editor for entering your instructions.

(2) A built-in wedge-like DOS manager invoked using either "@" or ">". This allows reading the disk directory at any time, reading the error channel at any time, and sending commands via the command channel #15 at any time.

(3) Meaningful error messages during execution. For example, if DOS reports a "FILE EXISTS" error, the message will be printed to the screen.

(4) As already mentioned, the ability to save your editor command files to disk (Put), and to retrieve them later (Get). The included examples take advantage of the Get command so you can load editor screens which we have prepared.

(5) The ability to load target routines to either their header address, or to a relocated address you specify. In either case, the ending address (plus one) is reported to you.

(6) The ablilty to generate a cross referenced symbol table, which is a list of all addresses referenced by the program, the assigned labels, and the addresses of all instructions which reference that particular address. It is an extremely powerful tool when you need it, as it allows you to instantly locate every call to any particular subroutine, and every reference to any particular variable, without having to search the code by hand. It is even quite useful in analyzing your own programs.

(7) The ability to import your own label names to be used in the disassembly.

3.Ø Introductory Comments

Symbol Master supports seven assemblers by generating compatible source code files and writing to disk:

1. MAE
2. PAL 64
3. Develop 64 (Machine Shop?)
4. CBM
5. LADS
6. Merlin 64
7. Panther

These are all C-64 assemblers, but presumably some will be converted to the C-128, and likely will have upwardly compatible source files. Since disk files are compatible between the C-64 and C-128, these assemblers can all be used to write programs for the C-128. The Symbol Master C-128 version was in fact written on a C-64 using MAE64 Version 5.Ø, and the resultant file then loaded into the C-128.

As assemblers specifically for the C-128 become available, we will likely add them to the C-128 version of Symbol Master, where available memory is not of particular concern.

Even if your assembler is not listed here, it is highly likely Symbol Master can be used with it. We suggest trying either the CBM-format file, which is a straightforward sequential file, or the PAL-format file, which is set up to load as a Basic program file (although it is not tokenized). Likely only slight editing, and perhaps changing some pseudo-ops, will be all that is required. Moreover, many assembler packages include conversion programs to convert from CBM-format source code files.

In each case below, the three letter name is the abbreviation used to specify that particular assembler to Symbol Master (on screen editor line 3). In most cases it is also the name of the assembler.

3.1 The Assemblers More Specifically

(1) MAE -- The MAE Macro Assembler/Text Editor. From Eastern House Software, 3239 Linda Drive Dr., Winston-Salem, N.C. 271Ø6. Telephone (919) 748-8446.

MAE is our personal favorite, despite its relative slowness compared to some of the others. For that reason, we have added MAE to the Schnedler Systems product line. Symbol Master and the PTD-651Ø Symbolic Debugger were both written using the MAE assembler, Version 5.Ø.

MAE64 Version 5.Ø is the only C64 assembler we are aware of which recognizes the additional mnemonics and addressing modes of the enhanced 65CØ2 instruction set, which Symbol Master Version 2.Ø optionally disassembles. The MSD disk drives, for example, use an enhanced version of the 65Ø2 which employs a subset of the full 65CØ2 instruction set.

(2) PAL -- PAL 64. From Pro-Line Software Ltd., 755 The Queensway East, Unit 8, Mississauga, Ontaria, L4Y-4C5. Telephone (416) 273-635Ø.

(3) D64 -- Develop-64. From French Silk, P.O. Box 7096, Minneapolis, MN 55407. Telephone (800) 328-0145, or (612) 871-4505 in Minnesota. This company has apparently been re-incarnated as FS! Software. Although we do not actually have one, we suspect this assembler is included in a product they are calling "Machine Shop". Symbol Master defaults to Develop-64 version 4.0 and above files. As an included option, the slightly different Develop-64 version 3.0 series source code files can be generated. If you have a version of Develop-64 below 4.0, it would definitely be worth it to check with them regarding an upgrade.

(4) CBM -- The Commodore 64 Macro Assembler Development System. Published by Commodore Business Machines, Inc., 1200 Wilson Drive, West Chester, PA 19380. Avaliable through Commodore software dealers.

(5) LAD -- The Commodore LADS Assembler, from "The Second Book of Machine Language", by Richard Mansfield, published by COMPUTE! Publications, Inc., P.O. Box 5406, Greensboro, NC 27403. Telephone (919) 275-9809.

(6) MER -- Merlin 64. Produced by Roger Wagner Publishing, Inc., 10761 Woodside Avenue, Suite E, Santee, CA 92071. Telephone (619) 562-3670.

(7) PAN -- Panther C64 Assembler. This was published by Panther Computer Corporation Corporation, 12021 Wilshire Boulevard, Los Angeles, CA 90025. Although we believe it is no longer available, those who have it seem to like it, as it is especially fast.

3.2 Relevant Consumer Information

Copy-protected assemblers: Develop 64 and Panther.

Unprotected assemblers: MAE, PAL, CBM, LADS and Merlin.

Beginning with Version 2.Ø, the Symbol Master package includes both: (1) A C-64 version which runs on the Commodore 64 computer, the "portable" version SX-64 (sometimes EX-64), and the Commodore-128 in its C-64 mode; and (2) a C-128 version which runs on the Commodore 128 computer in its native 128 mode. While the 4Ø-column screen editor was written first, it is our intention to complete the 8Ø-column screen editor for the C-128 version. Check the "LIST ME" file on the disk for current information.

A 1541, 1571 or compatible disk drive is required. With the C-128/1571 combination, the "fast" serial bus routines are fully operative. Since Symbol Master is not copy protected and uses only ordinary files, compatibility is highly unlkely to be a problem with any disk drive sold for use with Commodore computers. Device numbers 8, 9, 1Ø and 11 can be used for loading Symbol Master. While the initial default is device 8, for subsequent disk operations device numbers 8, 9, 1Ø and 11 can be used, and the current device number can be changed changed at will.

Selection of either Drive Ø (Ø:<filename>) or Drive 1 (1:<filename>) is supported and, indeed, we encourage use of this form. "Save with replace" (@Ø:<filename>) is supported for all disk file operations, if you dare use it. Recent magazine articles have advocated consistent use of the Drive number to avoid the "save with replace" bug in Commodore disk drives.

Dual disk drives are thus supported, and we ourselves use an MSD SD-2 dual disk drive.

Symbol Master is written with the expectation that the disk drive is connected in a normal manner to the serial bus. Symbol Master makes extensive use of the direct serial bus access routines, such as "LISTEN", "SECOND", "CIOUT", and the corresponding routines for input. Nevertheless, we have used Symbol Master on the C-64 with two different IEEE interface cards (not at the same time) in conjunction with the MSD disk drive, and experienced no problems. The two are the Bus Card II from Batteries Included, and the Quicksilver from Skyles Electric Works. (We have, however, experienced problems when using MICROMON with the IEEE interface cards.

Once Symbol Master is loaded its disk is no longer required, so your drive is fully available for the disk with the target program you are studying, or the disk to which you will be writing source code files.

A printer is optional, but highly recomended. The printer must be assigned Device #4 and likewise be connected in a normal manner to the serial bus. We have not tried any printers through the IEEE card. We have tested Symbol Master with both a Commodore 1525 printer (now MPS 8Ø1) connected directly to the serial bus, and a Centronics parallel input printer interfaced to the serial bus via a Cardco interface intended for that purpose. Both the Cardco "B" and the Cardco "G" work, and we would expect any interface which emulates a Commodore printer to work.

Symbol Master includes routines for reading target program files from cassette tape (Device #1). However, those routines have not been tested. In any event, you can read tape files in from Basic, and then enter Symbol Master via cold or warm start.

5.0 In General

Symbol Master in the C64 is a 24K system. However, 20K of this is under the Basic and Kernal ROMs, and under the I/O registers from $D000 through $DFFF. Thus only 4K is taken from normal program space. Approximately 12K is code and data tables. The other 12K is for variable arrays, including a 7500-byte label table generated during each disassembly (capacity 2500 labels) and a command file buffer.

The 7500-byte area for the label table is shared by user label names imported into the disassembler, and the 7500 bytes is the combined total available. Thus as longer lists of label names are read into the disassembler, the label capacity decreases. Each label name requires from three to nine bytes of memory, two bytes for the hex value, and from one to six bytes for the ASCII name. Each entry in Symbol Master's internal label table requires three bytes.

Symbol Master in the C64 includes two distinct portions, the editor and the main disassembler modules.

The editor portion of Symbol Master is provided in four separate versions, which are identical except for where they load. The respective address ranges, cold start addresses, and warm start addresses are as follows. The cold and warm start addresses are given both in decimal for use with the Basic SYS command, and in hex for use with the MICROMON .G command:

LOAD ADDRESS	COLD START ADDRESS	WARM START ADDRESS
$1000-$1FFF	SYS 4096 or .G 1000	SYS 4099 or .G 1003
$8000-$8FFF	SYS 32768 or .G 8000	SYS 32771 or .G 8003
$9000-$9FFF	SYS 36864 or .G 9000	SYS 36867 or .G 9003
$C000-$C000	SYS 49152 or .G C000	SYS 49155 or .G C000

In each case, the editor address range is implied by the filename. Thus "SM64EDIT$1.EXE" loads beginning at $1000, and is the first on the above list. The warm start address is always three more than the cold start address. Each Editor protects itself from Basic if necessary.

Rather than memorizing the SYS addresses above if you are entering from basic, the following equivalent can be used:

LOAD ADDRESS	COLD START ADDRESS	WARM START ADDRESS
$1000-$1FFF	SYS 4096	SYS 4096 + 3
$8000-$8FFF	SYS 4096 * 8	SYS 4096 * 8 + 3
$9000-$9FFF	SYS 4096 * 9	SYS 4096 * 9 + 3
$C000-$C000	SYS 4096 * 12	SYS 4096 * 12 + 3

The selected editor can be loaded by any one of three separate methods, at your option:

(1) The Basic direct-mode load command:
LOAD "0:<filename>",8,1. Follow this with a SYS to the appropriate cold start address. This procedure leaves Basic's pointers disturbed, so you will have to enter a NEW after the load to avoid a later "out of memory" error.

(2) Via the supplied boot program "BOOTSM64.BAS". The boot program will then automatically SYS to the appropriate cold start address.

(3) From the machine-language monitor: .L "Ø:<filename>". Follow this with a .G hhhh to the appropriate cold start address.

The editor will then automatically load the main modules. The main modules are the same regardless of the particular editor version selected. The file "SM64$A.EXE" loads under the Basic ROM beginning at $AØØØ. Following this module, but still under the Basic ROM, is the command file area, which extends nearly to $BFFF. The file "SM64$D.EXE" goes under the I/O, and extends partly under the Kernal ROM. Following that, up to $FFFØ, is a 75ØØ-byte area in which Symbol Master maintains its label table, shared by the imported label name list.

Not only does the editor load the main modules and provide for convenient entry of commands; it also drives the main disassembler. The main disassembler modules require the editor for support. They are not executable by themselves.

5.1 Zero-Page Memory Usage

Symbol Master makes extensive usage of zero page memory locations for its own purposes. However, in order to allow Symbol Master to be co-resident with other C64 programs, such as the machine-language monitor, programs in Basic, or an assembler, the entire zero page is saved to a swap area upon entry, and then restored upon exit from Symbol Master.

5.2 Other Memory Areas Used

Symbol Master uses essentially no RAM outside the ranges already mentioned above. In particular, the cassette buffer is not used, and the lower addresses on the Stack page 1 are not used. Symbol Master uses the stack only in the normal manner, for subroutine calls and as an occasional save location. Symbol Master does reset the stack pointer following entry.

Upon cold start entry, Symbol Master resets the Basic pointers to protect Symbol Master from Basic should you later exit to Basic. If you have not used the boot program "BOOTSM64.BAS", upon exiting to Basic for the first time, in direct mode you should enter NEW. A Basic "out of memory" error will otherwise likely result.

5.3 Symbol Master Environment

Symbol Master exists entirely independently of the Basic ROM, which in fact is banked out immediately when the Symbol Master editor is entered. (A normal exit from Symbol Master banks the Basic ROM back in.) Symbol Master does use several Kernal ROM routines, the documented ones where possible.

While the editor is active the Basic ROM is out, the Kernal ROM and I/O are in, and the IRQ interrupt is generally enabled. When the main disassembler is active, the Kernal and I/O are also banked out with interrupts off most of the time. When a line of output is ready to be generated, the Kernal, I/O and IRQ interrupts are momentarily turned on again.

Symbol Master changes the IRQ vector "CINV" at $Ø314,$Ø315, but restores this vector upon normal exit. This is an important aspect for full co-residency with a machine-language monitor, and other programs as well.

The NMI vector "NMINV" at $Ø318,$Ø319 is not disturbed at all by Symbol Master. The RESTORE key (which causes an NMI) should not be used while Symbol Master is running. An exit from Symbol Master via RUN/STOP/RESTORE will leave zero page quite scrambled, with little chance of recovery.

5.4 Co-residency with Other Programs

As implied above, Symbol Master can be co-resident with any program which does not use the same memory. Moreover, Symbol Master can be exited and re-entered via the warm start address at will, with no loss of your command screen. This of course is a great convenience because it avoids the need to repeatedly Load in programs as you go from one to the other.

Most notably, Symbol Master can be co-resident with a machine language monitor, such as the supplied MICROMON.

Symbol Master can be co-resident with either the PAL assembler, the MAE assembler (MAE Version 3.Ø easily, and MAE Version 5.Ø if you use care), or the LADS assembler. It works best to first load and cold start Symbol Master, exit to Basic, enter NEW, and then load the assembler. Re-enter Symbol Master via the warm-start SYS address. To go from Symbol Master to PAL, just exit to Basic, since PAL exists in the BASIC environment. To go from Symbol Master to MAE, first exit to Basic and then SYS to either the MAE cold start address $5ØØØ (SYS 2Ø48Ø) or the MAE warm start address $5ØØ3 (SYS 2Ø483). Alternatively, you can exit to and go through the machine-language monitor.

Symbol Master at $CØØØ can probably be co-resident with the CBM assembler, but not the CBM editor, if you would want to try it. Symbol Master at $9ØØØ can probably be co-resident with the CBM editor, and perhaps with the ASSEMBLER64.

While file-compatible with Develop-64, Symbol Master cannot be co-resident with Develop-64 because Develop-64 also uses the RAM under the Basic ROM. Also, Develop-64 is too heavily protected to be easily exited and re-entered without reloading.

While we have not tried them, we suspect a similiar conflict would exist with Merlin or Panther and Symbol Master in memory at the same time.

Symbol Master in the C64 is compatible with its own label name editor (which is a separate program) in memory at the same time, but Symbol Master should be loaded at $8ØØØ, $9ØØØ or $CØØØ, since the label name editor always loads at $Ø8Ø1. Symbol Master and its label name editor communicate with each other only through disk files.

6.Ø In General

The Symbol Master file structure in the C128 (128 mode) is much more straightforward because each of the two Symbol Master versions (for RAM Bank Ø or RAM Bank 1) is self-contained as a single load file. Either of the versions can disassemble from any of the 16 logical Bank configurations $Ø through $F which Commodore has defined.

Symbol Master in the C128 is not intended to be co-resident with programs in Basic, although Basic direct-mode commands can still be used. It is compatible with the built-in Monitor in the C128, and you can freely pass between the two.

Symbol Master in the C128, unlike the C64 version, makes no attempt to protect itself from Basic by changing Basic's pointers. Accordingly, more care must be used to avoid clobbering the Symbol Master code. Particularly in the Bank 1 version, avoid creating any Basic variables. Even in Basic direct mode, a syntax error seems to cause memory in RAM Bank 1 to be altered beginning at $Ø4ØØ, overwriting Symbol Master code.

6.1 RAM Bank Ø Version

The file "SM128/BANKØ" is LOADed and RUN like a Basic program. If you LIST it, you will see a one-line Basic program with a SYS to the cold-start address 7186 decimal, which is $1C12 hex. The corresponding warm start address is 7189 decimal, which is $1C15 hex, and can be re-entered either by SYS 7189 from Basic, or G Ø1C15 from the Monitor.

To Load from Basic direct mode:

DLOAD "SM128/BANKØ", followed by RUN

Since Symbol Master in Bank Ø loads to the start of the Basic text area, be certain you do not invoke the Basic GRAPHIC command which shifts the start of Basic.

The Bank Ø version loads ending at approximately $4FFF in Bank Ø. Following that are program variables, followed in turn by a vast area, in excess of 4Ø,ØØØ bytes, for the label names and label table. Symbol Master will use all of RAM Bank Ø up to $FEFF, if necessary. However, it always works upward, so it is probably safe to use the higher areas of memory.

6.2 RAM Bank 1 Version

Loading of the file "SM128/BANK1" is straightforward from Basic direct mode:

BLOAD "SM128/BANK1",B1

The cold start address, in Bank 1 is $Ø4ØØ, which is decimal 1Ø24. The warm start address is $Ø4Ø3, which is decimal 1Ø27. The Bank 1 version is longer than the Bank Ø version, but it starts at a lower address, and leaves even more memory free at the end, in this case in RAM Bank 1.

To cold start from Basic direct mode:

BANK 1: SYS 1Ø24.

To subsequently warm start from the Monitor: G 1Ø4Ø3

CHAPTER 7: INTRODUCTION TO THE SYMBOL MASTER COMMAND SCREEN EDITOR

The command screen editor is one of the things which makes Symbol Master so easy to use. This chapter briefly introduces the editor and the various lines on it. The following chapters treat individual lines and areas of the editor screen in detail.

The C64 and C128 versions are nearly identical. There is one difference, and that is the C128 version requires you to select the logical bank configuration to disassemble from. Details of this are given in Chapter 9, which describes screen editor line 3 in detail. The present Chapter 7 applies to both the C64 and C128 versions.

There is another screen editor included as a separate program. That is the Label Name Editor covered in Chapter 13, and is used to generate and modify label name files to use in the disassembly.

7.1 Overall Guidelines

Here are a few things to have in mind at the outset:

(1) The editor is a full screen editor, with full cursor movement, and entry of any line by pressing RETURN with the cursor anywhere on that line.

(2) Line 5 is a divider line. The cursor will move freely from above the divider line to below, but may appear to be trapped below the line. The reason is that the entire screen below the divider line becomes an up and down scrolling window when there are more than 20 block lines below the divider line.

To get the cursor back above the line, simply use the HOME key. An alternative is to use SHIFT/CLR, followed by RETURN. Another alternative is to put the cursor on any blank line and press RETURN.

(3) The editor can be re-initialized at any time with no loss of entered information by pressing RETURN with the cursor on any blank line. A quicker way sometimes is to clear the screen with SHIFT/CLR, and then RETURN. Re-initializing the editor has the effect of filling the screen with your entered instructions as they have been interpreted and stored by the editor. To be sure everything has been entered correctly, it is a good idea to re-initialize the editor and review the screen before Running the disassembler with the "R" command.

(4) As you enter each line, the following process occurs: The screen line is scanned from left to right, while the editor interprets and stores the information. If a syntax error is encountered, scanning stops, and a question mark (?) is printed on the line following the point where the syntax error was encountered. Following a successful scan, the screen line is cleared entirely, then rewritten from memory. You will see the line flash. This is your assurance that your entered line has been accepted.

(5) Lines are limited to 39 characters in length, and all the syntaxes are defined so that this limitation need not be exceeded. If you do, the editor may become confused, and have to be reinitialized as described in (3) above.

(6) Sometimes, following a syntax error, the editor will refuse to accept the line even after you retype it correctly. The reason is that scanning is beginning at some intermediate point on the screen line. The solution is to

reinitialize as described in (3) above. We believe, however, that this has been largely corrected in Symbol Master Version 2.Ø.

(7) Upon warm start to Symbol Master, the editor screen you had on last exiting will still be there.

(8) Caution: Do not use the RESTORE key. The RUN/STOP/RESTORE combination will attempt to send you to Basic, but the Basic ROM will not be in and zero-page will be messed up. Recovery is unlikely. There are three proper ways to exit the Symbol Master editor, use them: "B" -- Break to monitor; "X" -- eXit to basic; and "K" -- Kill, cold start the computer.

7.2 Editor Screen Line Summary

The individual editor lines are described in detail in the next four chapters of this documentation. Here is a brief list:

Line 1 -- (Top line) Command input line. All direct mode commands for immediate execution are entered on this line.

Line 2 -- Output line. Symbol Master output messages, including DOS command channel messages, are output to this line. Inputs are not accepted.

Line 3 -- Miscellaneous parameter line. Here is where you enter several parameters relating to the program you are disassembling.

Line 4 -- Control and Equate file name line.

Line 5 -- Divider line. Inputs are not accepted.

Lines 6-25 -- Block lines. Here is where you enter parameters telling Symbol Master where to find the code to disassemble, its type, and filenames to assign if writing to disk. These lines are the scrolling window, which you will see when there are more than twenty blocks.

7.3 Scope of Command File

All of the information on editor screen lines 3 through 25 is part of the command file which is stored in encoded form under the Basic ROM (in the C64). This file can be saved to disk using the "P" (Put) command on line 1, and later retrieved using the "G" (Get) command. Command files can also be combined using the "A" (Append) command.

The disassembly examples we have included use this feature. You can Get them, and avoid typing.

Commands you enter on editor screen line 1 are transient, and are not part of the command file.

We have adopted the filename extension ".CMD" to designate command files Put to disk. This is not a requirement, though, and a command file name can be anything, except embedded spaces are not permitted.

Command files are written to disk as SEQuential files.

Command files are completely compatible between the C64 and C128 versions of Symbol Master. Command files created with earlier Symbol Master versions 1.Ø, 1.1 and 1.2 are upwardly compatible, but not the converse: You cannot create a command file with version 2.Ø and go back to your version 1.Ø Symbol Master, in case you have upgraded.

Each command is started with a single letter or character from the list below as the first character on the line. Do not use more than one character to indicate the command. I.e., for Get use just G, not GET.

Some of the commands have further parameters, and these follow the command letter or character. A space between the command letter or character and the further parameters is optional.

They all have the same syntax and effect for both the C64 and C128 versions, with the exception of the "L" (Load) command. The C128 requires the RAM Bank to be specified,.

The commands are listed first, and then each is described:

A -- Append command file.
B -- Do a break, usually to monitor.
C -- Clear command file in memory.
F -- Fix Column 2 or 3.
G -- Get command file from disk.
H -- Print a Hardcopy of the command file on printer device #4.
K -- Kill. Cold start the computer.
L -- Load target routine.
M -- Memory bytes left.
N -- Null label name list in memory.
P -- Put command file to disk.
R -- Run the disassembler.
U -- Read in User label name file.
X -- eXit to Basic.
@ -- DOS command.
> -- DOS command. Same as @.
-- Change current disk device number.

A -- Append Command file

Syntax:

```
A "Ø:<filename>"
      or
A Ø:<filename>
      or
AØ:<filename>
```

Note: <> enclose required parameters, but the <> are not themselves entered.

Similiar to Get, below, except retains the previous command file on the screen, appending to the end. The header information (everything above divider line 5) from the file being read in is ignored, and the information originally on Lines 3 and 4 remains, except the block count (first entry on Line 3) is updated. If the new block count would exceed 99, the append is not done, an overflow message is generated, and the original information on the screen remains unharmed. The total blocks there would have been if the overflow had not been trapped (a number greater than 99) is reported.

B -- Do a Break (to monitor)

There are no parameters.

Zero page and the IRQ vector are restored to what they were upon cold or warm start entry to Symbol Master. The Basic and Kernal ROMs are banked in (C64), or the system configuration is restored (C128). IRQ interrupts are enabled. A break instruction is executed. In the C64, if a machine-language monitor is in place and has been initialized, it will be entered. Otherwise Basic will be entered via the RESTORE entry. (Warning: Never use the RESTORE key itself when in Symbol Master). In the C128, the Monitor will be entered.

C -- Clear Command File

There are no parameters.

Clears the command file in memory to the cold start default.

F -- Fix Column 2 or 3

Syntax:

```
F2
  or
F3
  or
F3 hhhh
```

Saves a lot of entering numbers by hand by automatically generating columns 2 and 3 of the memory block definition lines. I can't believe I didn't think of this earlier to include in Version 1.Ø.

The detailed explanation of the "Fix" command is given in Chapter 11 concerning the memory block definition lines, because that is is the context in which it will make sense.

G -- Get Command File

Syntax:

```
G "Ø:<filename>"
    or
G <filename>
    or
G<filename>
```

Note: Again, the <> enclose the required parameters, but are not actually entered.

Gets a new command file from disk and re-initializes the screen with the new file. Anything previously on the screen is overwritten. See above for the append function (A).

H -- Hardcopy of Command file

There are no parameters.

Dumps the entire command file to printer device #4, starting with screen line 3.

K -- Kill, Cold Start via Software Reset.

There are no parameters. However, since this is a rather drastic action, you are prompted whether you are sure. Key "Y" if you mean it. Any other key reinitializes the editor.

Does a jump through the hardware reset vector at $FFFC,$FFFD to $FCE2 (C64) or $FF3D (C128). This cold starts the computer, and preserves most memory contents. Primary use in the C64 is to save wear and tear on the ON/OFF switch. In the C128 in the C64 mode the C64 mode remains after the cold start, so this command is quite different from the action of the reset button.

You would not normally re-enter Symbol Master after doing this, but a warm start will still be possible, provided Symbol Master has not been overwritten. Although Symbol Master changes the Basic memory pointers to protect itself (C64 only), these pointers are all reset by a cold start of the computer, leaving Symbol Master vulnerable to Basic (unless loaded at $C000).

L -- Load Target Routine

This is a very important command which you will normally use to load in your target programs to be disassembled and studied. If you prefer, the target routine can be loaded via the MICROMON machine-language moditor, and then Going to the Symbol Master warm start address.

Before using this command you should know fairly well where the target routine will load, how long it is, and where the blocks of code and tables are. Use the specially-enhanced MICROMON for these investigations (C64 or C64 mode only).

Syntax for C64 version:

L "0:<file name>" Note: Do not actually enter the <>.

Loads a PRG file from disk into memory doing a non-relocated load to the starting address specified by the two-byte program file header. The ending address (plus one) is reported to you in hex.

L hhhh "0:<file name>"

Loads a PRG file from disk into memory doing a relocated load to the starting address hhhh. The program file header is ignored. The address hhhh must be a valid hex address. A "$" prefix is not used. All four places must be given, even if the first is 0. The ending address (plus one) is reported to you in hex.

Note: Sometimes it is necessary to press RETURN twice before the command will enter. We don't know why.

The filename must have quotes around it. Note that the rule is different for files you tell Symbol Master to generate. Embedded spaces are permitted. Also, the above assumes that the Drive number "0" is being employed as part of the filename. You do not need to do this, but it is best to use it. Drive "1" can of course be specified instead.

We have included the code for doing loads from tape, but this has not been tested as this is being written. The intended syntax for a load from tape is:

```
L "<file name>" 1
     or
L hhhh "<file name>" 1
```

We have a particular reason for questioning whether tape load will work. Symbol Master at present uses the regular Kernal LOAD routine for these loads. Although it is said that the LOAD routine will do either a header address load or a relocated load, it seems that files saved to tape with a secondary address of "1" will load to their header address even if you ask for relocated.

Syntax for C128 version

```
Lr "Ø:<file name"                    (non-relocated load)
     or
Lr hhhh "Ø:<file name>"                  (relocated load)
```

Where r is Ø, 1, 2 or 3 to specify the RAM bank into which the load is to be done. In a C128 only RAM Banks Ø and 1 are present, and 2 and 3 are future expansion. In a 128K C128 RAM Bank 2 is the same as Ø, and 3 is the same as 1. Normally when you have "SM128/BANKØ" loaded you will load target code into RAM Bank 1, and when you have "SM128/BANK1" loaded you will load target code into RAM Bank Ø. However, no restrictions are placed on the command, and it is up to you not to do a load which would overwrite the Symbol Master code.

M -- Memory Bytes Remaining

There are no parameters.

Gives you the amount of memory left (in view of the length of an imported label name file) for Symbol Master's generated label table.

N -- Null Label Name List in Memory

There are no parameters.

Restores maximum available memory for the generated label table. Probably will never be necessary, and certainly not in the C128.

P -- Put Command File to Disk

Syntax:

```
P "Ø:<filename>"              Note:  Again, <> enclose the
    or                               required parameters, but
P Ø:<filename>                       not actually entered.
    or
PØ:<filename>
```

This is the converse of Get. Saves the command file to disk as a SEQuential type. What is on the screen remains.

The Command files are compatible between the C64 and C128 versions. You can Put from one, then Get into the other. (Also, as already noted, command files are upwardly compatible from earlier versions of Symbol Master.)

R -- Run the Disassembler

Syntax:

R [options]

This is the most important command because nothing would happen without it. It passes parameters and control from the editor to the main disassembler, which then executes. At the conclusion of the disassembly, you are returned to the warm start point, and are reminded which version of Symbol Master you have loaded.

The options are C, X, S, D, P and F, described below. They may be in any order, with or without spaces. If inconsistent options are used on the same line, the last encountered on the line will control. It is not necessary to specify any options, as there is a default case, defined below.

C -- output Code and tables.

The disassembler output, wherever directed, will include a list of equates to external label references, and a disassembly of the blocks of code and tables defined on the block definition lines of the command screen. This is the fundamental disassembly.

X -- generate cross-referenced (Xr) label table.

On output to screen or printer, a cross-referenced label table will be generated following the code and tables. The X option may be selected without C. Unless you use the CBM assembler (which can itself generate a cross-referenced label table), you may want to disassemble your own programs just to get the cross-reference generated by Symbol Master! If you direct output to disk, the X option is ignored.

S -- direct output to Screen.

D -- direct output to Disk as assembler compatible source code files.

P -- direct output to Printer, device #4.

F -- if printer output selected, format the output with one-inch (6 line) page breaks every 60 lines.

Here are two defaults intentionally provided:

R

Same as selecting options C X S.

R D

Same as selecting options C D.

The above two are the only defaults intentionally provided. In general, once you start specifying options, you must specify everything you intend. For example, "R P" will not generate any output at all since you have not specified either "C" or "X". Examples of correct commands are "R PC", "R PCX" and "R PX".

U -- Read User Label Name File from Disk

Symbol Master Version 2.Ø (unlike 1.Ø) does not have in it when cold started any label names for use in the disassembly. The "U" command is used to load them in. While this approach requires an extra step in loading, the advantage is much greater flexibility. The nature of a label name file is discussed in Chapter 15. The Label Name Editor can be used to create your own or modifiy the ones we have provided. A provided label name file which is nearly identical to the label names which were permanently built in to Symbol Master Version 1.Ø thru 1.2 is included and named "DEFAULT64.LBL".

The syntax is:

```
U "Ø:<filename>"
     or
U Ø:<filename>
     or
UØ:<filename>
```

Note: Again, <> enclose the required parameters, but are not actually entered.

To avoid confusion between Command files and Label Name files, Command files are always file type SEQ and Label Name files are always file type USR. The "G" and "U" commands check for the proper file type, and report an error if the wrong one is used. We use the respective file name extensions .CMD and .LBL as a matter of convenience, but there is no requirement. The filenames can be anything you like.

X -- eXit to Basic

There are no parameters.

Zero page and the IRQ vector are restored to what they were upon cold or warm start entry to Symbol Master. The Basic and Kernal ROMs are banked in (C64) or the system configuration is restored (C128). A JMP through the Basic warm start vector at $AØØ2,$AØØ3 to $E37B is done (C64), or the vector at $ØAØØ,$ØAØØ to $4ØØ3 (C128). This is very similiar to the RESTORE entry. (Warning: Never use the RESTORE key itself when in Symbol Master).

@ and > -- DOS Commands

These work just like a DOS wedge. See Chapter 12 for a more detailed description.

Note: If you enter >$Ø to read the disk directory, the listing can be paused and restarted using the spacebar. The STOP key stops the directory listing. After a directory listing, RETURN gets you back to the editor screen.

-- Change Disk Device Number

```
# n
```

Where n is 8, 9, 1Ø or 11 will change the disk device number Symbol Master uses for its disk operations, including DOS commands. The default on cold start is 8, regardless of the device number used to load the disassembler.

Screen line 3 is the first line which affects the command file. Syntax for the C64 version is as follows:

<nn> <f> <p> <asm> [OPT] Note: The <> enclose the parameters, and are not actually entered.

Where --

<nn> is the number of blocks of code and tables, in decimal. The allowable range is 1 through 99. On cold start default, the number is 1. To see its effect, change it to anything in the range 2-99. Enter the line, and then re-initialize the editor by pressing RETURN with the cursor on a blank line. (This number must be set and entered before you can add blocks to screen lines 6-25.)

<f> is the single or multiple file control. It must either be S (Single) or M (Multiple). If output is directed to disk with the S option set, a single file will be written to disk, and its name will be the "controlfilename" entered on editor screen line 4, as described in the next section. If output is directed to disk with the M option set, multiple files will be written, including "controlfilename", "equatefilename", and at least one "modulefilename". S and M have no effect on output to the screen or printer, but the editor still requires one or the other to be present.

<p> is the label processing control, and determines which labels are converted to names. There are three possibilities, N (None), K (Kernal), and A (All). The effect of K and A is described in the separate section on Operating System Name labels. If N is selected, only arbitrary labels are generated, as described in detail in the Chapter 14.

<asm> is the name of the assembler for which disk files will be generated, and the syntax of which will be used for pseudo-ops. For familiar pseudo-ops, set it for your assembler, even if you are not writing files to disk. (You may however not want to select LADS during trial runs which are not written to disk because LADS does not support the .WORD pseudo-op, and thus "WO" and "W-" blocks discussed in Chapter 11 are not analyzed when LADS is selected.) The allowed options are MAE, PAL, D64, CBM, LAD, MER and PAN. Only enough letters to uniquely identify the assembler need be entered. Thus "C" as the first character will give you "CBM", while "PAL" and "PAN" must have all three entered.

OPT, if present, indicates to invoke an OPTion relating to the particular assembler selected. The options are quite different depending on the assembler. There is no general statement which can be made. Refer to the Chapter on Specific Assembler Notes. If you want to clear OPT, spacebar or DELete over it, and press RETURN to enter the line.

The syntax for the C128 version has one additional parameter, the bank parameter "b":

<nn> <b> <f> <p> <asm> [OPT]

b is a single hex character in the range Ø through F, and specifies the logical bank configuration number to disassemble from, with definitions per the official Commodore documentation. While all 16 are available, there are three which are of primary interest: Ø for RAM Bank Ø, 1 for RAM Bank 1, and F for the System Bank, which includes all the ROMs ($4ØØØ through $FFFF), and the remainder $ØØØØ through $3FFF RAM Bank Ø. Symbol Master sets this parameter to F by default.

CHAPTER 10: SCREEN EDITOR LINE 4
CONTROL AND EQUATE FILENAMES LINE

Screen line 4 is for the file name for the control file to be written to disk, and for the file name for the equate file to be written to disk. The syntax is:

[0:<controlfilename>] [0:<equatefilename>]

Each can be up to eighteen characters in length, to allow for the disk syntax "0:filename" or "1:filename". Quotation marks around the names are not permitted. Embedded spaces are not permitted because the editor considers the first space to be the end of the control file name, and the second space to be the end of the equate file name.

Again, in the definitions above the <> merely enclose the parameters, and the <> themselves are not entered. The [] indicate these parameters are optional, unless disk output is selected. The [] are not actually entered either.

On disassembler output to screen and printer, these names are entirely optional, and in fact are ignored.

On disassembler output to disk, the control filename is always required. If the S (Single) file option is selected on Line 3, the single file will be written with the control filename. No other filenames are required.

If the M (Multiple) file option is selected on Line 3 with output to disk, both the control filename and the equate filename are required. Also, at least the first memory block line below the Line 5 divider must have a filename. When Symbol Master is Run, a set of files will be generated appropriate for your assembler. On MAE, D64, CBM, MERlin and PANther output, the control file will refer to the label file and each module file in order. The pseudo-op for MAE is .FI. For D64 it is LIB. For CBM it is .LIB. For MERlin it is PUT. For PANther it is LOA. On PAL and LADS output, the control file will chain to the equate file using the .FILE pseudo-op, the equate file will chain to the first module file, and so on. The last module file then chains back to the control file.

In any case, start your assembler with the control file, and it will take care of the rest.

If you happen to trust the "save-with-replace" command on your disk drive, you can use this form with the control filename, the equate filename, as well as the module filenames, yet to be described. The syntax is @0:filename. It is your decision whether to use this form. By now the magazines, primarily Compute's Gazette and Transactor, have demonstrated beyond all doubt that the long-reported bug in the DOS "save with replace" command is in fact real. It has also been suggested that consistent use of the drive parameter 0:, even if you have a single disk drive, will at least minimize problems. I will report that I use it constantly with the MSD dual drive, which has never given me an error.

11.Ø Formal Syntax Definition

These lines tell Symbol Master where to find the code and data tables to be disassembled, and how to treat them. The actual number of these lines is determined by the first number on screen Line 3, which is for miscellaneous parameters. The minimum is 1, and the maximum is 99.

Here is the syntax for each line:

<nn> <t><i> <1111> <2222> <3333> [Ø:<filename>]

Note: The "t" and "i" parameters are always run together as a two-character pair, and the "i" parameter is not present at all when the "t" parameter is "C" (for Code). Hopefully the description below will not be too confusing. The filename is optional, required only when writing source code files to disk, and even then is not always required. Again, the <> merely enclose the parameters for purposes of definition, and are not actually entered.

Where --

<nn> is the block number in decimal. The blocks should be numbered in sequence.

<t> is the type of the block. Primarily, whether Code or Data tables. "C" means code, i.e., ordinary instructions. "T" means table data which you want to be formatted three bytes per line on the disassembly. "W" means word table data which you want to be formatted two bytes per line on the disassembly. A "C", "T" or "W" is always required.

Here is a confusing one to define, and one where a significant enhancement has been added to Symbol Master Version 2.Ø:

<i> has two completely different uses. The "i" parameter is not required and not allowed following a "C". It is mandatory following "T" or "W".:

(1) Controls the interpretation of ASCII bytes. For this, the parameter is "A" or "N". "A" means to intrepret bytes as ASCII data in the comment field. "N" means no ASCII intrepretation. See the note on ASCII intrepretation below. For this purpose there are four possible combinations for the parameter pair <t><i>: WN, WA, TN and TA.

(2) Defines a table of .WORD pairs, also known as address constants, and indicates whether they are of the form LABEL or LABEL-1. For this, the parameter is "O" (as in wOrd) or "-" (as in LABEL-1). "O" means to treat the two-byte table entries as address constants in low-byte-first order, and to generate a label accordingly referencing the address thus defined. "-" is similiar, but assumes the table was generated during assembly using entries of the form .WORD LABEL-1, as is common in 65Ø2 programming where a table of action addresses is set up. These entries are usually pushed onto the stack, and then reached via an RTS. For this second purpose there are two possible combinations, for the parameter pair <t><i>: WO and W-.

(2a) Further note on .WORD pair tables: When initially studying a target program to disassemble, a .WORD pair table can often be recognized because it makes no sense interpreted as ASCII, and there is a pattern to the high bytes in that they fall in the same general range, almost always within the range of the program. The .WORD pairs may point to either data, or to routine entry points. By doing trial disassemblies with both "WO" and "W-" it will readily be apparent which form to use. The right one will point to addresses of bytes which begin a line of source code, while the wrong one will be referencing the second or third bytes of instructions, and generating comments accordingly.

(2b) Still further note on .WORD pair tables: The LADS assembler does not support the .WORD pseudo-op. When LADS format is selected, "WO" and "W-" and both treated simply as "WN". Otherwise the generated source code files would not work in the assembler. Obviously this does not give you a very good analysis, and for this reason we recommend you do not select LADS format during your initial study when you are outputting to screen and printer. PAL is a good alternative format. When you are finally ready to write a file to disk, then switch to the LADS format, and plan on commenting your source code by hand to explain the resultant .BYTE pseudo-ops.

(2c) Final note on word pair tables: Here are the .WORD pseudo-ops appropriate to each of the assemblers supported (other than LADS). For MAE it is .SE (you might want to change to .SI). For PAL and CBM it is .WORD. For MERlin it is DA (define address). For PANther it is ADR. Develop 64 uses BYT, and is able to recognize that low-byte-first order is what is intended.

<1111> and <2222> are the starting and ending addresses which define the block where it actually resides in memory for disassembly. <1111> is not necessarily the origin address, but it can be. If <1111> is not the origin address, then Symbol Master automatically applies an offset for proper disassembly. <1111> and <222>2 are both given in hex, as four hex bytes. A "$" prefix is not used, and all four places must be specified.

<3333> is the origin address, also in hex and also having four places. If the code is loaded into memory where it normally runs, <1111> and <3333> will be the same. As a converse example, you may be disassembling code which loads and runs at $C000 through $C2FF, but for purposes of example you have loaded the symbol master editor there. The choice is yours, but you might want to load the target program at $2000. Under these circumstances, for <1111> <2222> <3333> you would enter 2000 22FF C000.

<filename> is the filename for the block. It is entirely optional for output to screen or printer. For output to disk, only block 1 needs a filename. If present, the filename is included as a comment in the generated source code.

11.1 Note on Effect of Block Filenames

The block filenames have a significant effect when writing multiple files to disk, because the presence of a filename triggers the closing of a previous file, and the opening of a new one. This approach offers a significant advantage when writing module files: If you have short blocks of code and tables in a program you can switch back and forth by defining successive blocks, but you won't be bothered by having a corresponding bunch of short disk files. Instead, you can cause a new disk file to begin when a reasonable amount of memory has been disassembled.

11.2 Note on Relationship of Block Memory Ranges to Each Other

There is no requirement that the memory ranges from one block to another be contiguous, or even in any particular order. You will probably want to keep them in order for meaningful results, but there is no requirement.

Symbol Master issues an appropriate ORIGIN pseudo-op at the beginning of disassembly of Block 1. It is commented ";INITIAL ORIGIN". As each subsequent block is processed, it is first checked to see whether it picks up where the previous one left off. If so, output simply continues. If not, a new ORIGIN pseudo-op is issued, and commented ";NEW ORIGIN".

Do not overlap the address ranges of the memory blocks. If you do, duplicate labels will be generated, which will cause your assembler to issue an error message later. It is also possible that Symbol Master will report an execution error, with abort and warm start.

11.3 The "Fix" Commands

Chapter 8 mentions the Fix commands F2 and F3, but saved the explanation to this chapter. These two commands save you the burden, in most cases, of typing columns 2 and 3 of the addresses, i.e., the <2222> and <3333> parameters.

In most cases the blocks of memory ranges you are disassembling are contiguous, which means that the <2222> parameter of each block line is just one less than the <1111> parameter of the next block line. The F2 command does this subtraction for you, and enters the whole column, with the exception of the very last entry, since there is no "next" block to subtract from. Even if there are a few which are not contiguous, it is easier to use the F2 command to fix most of the entries, and then edit the few remaining ones.

Caution: When using the F2 command, be certain you enter the last <2222> parameter yourself. Otherwise it may be left as ØØØØ, and you will be sending the disassembler on a very long trip which will not end until it has either wrapped around $FFFF to $ØØØØ, the label table has overflowed, or you get an execution error by attempting to disassemble through an area of memory which is changing.

Similiarly, in most cases the <3333> parameter is either the same as <1111>, or all the <3333> parameters have a fixed offset from the <1111> parameters. The command F3 alone copies all the <1111> entries to the <3333> on the same line. The command F3 <hhhh> adds a hex offset. Since two-byte addition is performed, you can in effect subtract by adding a large enough hex value to wrap around $FFFF. Rather than trying to determine the correct <hhhh> in advance, it is usually easier to use trial and error a few times.

To conclude this subject, in most cases you can: (1) Go down the first column entering all the block starting address parameters <1111>. (2) Enter the last <2222> parameter. (3) Use F2 to fix the remaining <2222> parameters. (4) Use F3 [<hhhh>] to fix all the the <3333> parameters.

11.4 Tip on Entering Lines

You cannot enter a block line number greater than the maximum defined at the beginning of editor Line 3. You can go back and increase the number at any time. While you could type these in yourself in order on a blank screen, it is easier to re-initialize the screen to print the numbers, and to then type over the default which appears.

11.5 Inserting and Deleting

There is no insert and delete function. However, you can move lines to prepare for an insert or delete quite easily by typing new line numbers over existing lines, and entering with RETURN. Lines need not be entered in order. As each line is entered, it is scanned and the line number is picked up.

Don't forget the "A" (Append) command from Chapter 8 which does a useful and related function.

Do not leave unused block lines at the end. The list can be easily truncated by changing the number of lines specified as a parameter on editor line 3. This does not delete the information from memory; it merely changes a pointer. The information will still be there if you later increase the number of lines parameter.

11.6 ASCII Intrepretation

When ASCII interpretation for a block is selected (WA or TA), the results are put in the comment field, not the actual BYTE pseudo-op field. There are several different ways ASCII data can be placed in programs. For example, there are screen codes in the range $00 through $1F, regular ASCII in the range $20 through $5F, and uppercase ASCII in the range $60 though $7F. In addition, bit 7 might be set, with the whole pattern beginning over at $80. To increase your chances of being able to recognize ASCII text data regardless of range, all bytes to be intrepreted are mapped into the range of upper case printable CBM ASCII values $20 thru $5F, and the result is put in the comment field.

You may eventually want to edit the disassembled source code to include ASCII text strings in the BYTE pseudo-op field, using of course the appropriate pseudo-op for your assembler. Do this with caution. Since there are at least six ranges within which ASCII characters may be found, simply taking the interpreted ASCII data from the comment field and moving it to the BYTE field will not necessarily recreate the original.

CHAPTER 12: DOS COMMANDS

This Chapter is not specific to Symbol Master, because the commands summarized below are all interpreted and executed by the Disk Operating System within the disk drive. All the DOS manager included in Symbol Master does in this regard is send and receive. This section however is written to make it clear what can be done with the Symbol Master DOS manager, which is included in both the C64 and C128 versions.

In general, the Symbol Master DOS manager behaves exactly like a DOS "wedge" in the C64, except non-relevant commmands are not provided. The not-provided non-relevant commands are the ones which Load and Save Basic programs. Also, machiné-language load (%filename in the normal wedge) is not provided because the same command is provided in an enhanced manner by the Symbol Master "L" (Load) command, which optionally allows either a relocated load or a header load.

The screen editor top line is used for the DOS manager. To invoke the DOS manager, the first non-space character on the line should be either ">" or "@". The two produce precisely the same results. In the examples below the notation ">" only is used to avoid duplication. If you prefer, read "@" wherever you see ">".

Here are the primary DOS commands:

>	Read the disk error (command) channel
>$Ø	Read the disk directory. Note, spacebar pauses and restarts. STOP aborts. Return to editor command screen with RETURN. Inclusion of the "Ø" after "$" is optional, but is recommended even on a single disk drive to minimize the "save-with-replace" bug Omitting the "Ø" on a dual drive reads both disk directories.
>$1	Read the disk directory for drive 1.
>NØ:diskname,id	Format a disk. (Long New).
>NØ:diskname	Clear disk directory. (Short New).
>IØ	Read BAM to initialize drive.
>VØ	Validate disk. Use whenever a file is left unclosed, as indicated by an * next to the file type. Never scratch such files.
>RØ:newname=Ø:oldname	Rename a file.
>SØ:filename	Scratch (delete) a file

The above is of course not a complete list, but does include the more important ones, and will serve to illustrate the capability.

CHAPTER 13: SYMBOL MASTER FILENAMES

This Chapter, for the sake of clarity, repeats and summarizes imformation which is perhaps only implied in other Chapters detailing the various commands.

The Symbol Master editor limits all filenames to eighteen characters. This allows for filenames of the form "Ø:filename", which is required with a dual disk drive, and a good idea even with a single disk drive.

In general, the Symbol Master editor uses spaces as deliminators between fields. In order to save space on the screeh so that lines need not exceed 39 characters, quotation marks (") are not used around the filenames which Symbol Master is itself being instructed to write to disk. The rule is this: embedded spaces are not permitted in the filenames which Symbol Master is being asked to write to disk.

The above limitation applies to the Control File name, the Equate File name, and each Module file name. In addition, the limitation of no embedded spaces applies to Command file names which are written via the "P" (Put) command, and retrieved via the "G" (Get) command; as well as to Label Name filenames which are written via the "W" command (Label Name Editor only, see Chapter 15), and retrieved via the "U" command.

However, embedded spaces are permitted in files which Symbol Master does not itself necessarily generate. There are two situations in particular where this applies.

The first is the case of target program files to be disassembled. Such files are loaded by either the command (in C64 version syntax):

```
L "Ø:<file name>"
     or
L llll "Ø:<file name>"
```

Here, quotation marks are required, and embedded spaces are permitted.

The second situation is when the DOS manager is invoked via "@" or ">". Operation is exactly like a DOS "wedge". Quotation marks are not used around filenames, and embedded spaces are permitted.

With the Symbol Master "G" (Get), "P" (Put) and "U" commands, quotation marks around the file name are optional, to avoid confusion with the "L" (Load) command which has similar syntax. So, while not needed, there is no harm in using quotation marks with "G", "P" and "U". (Do not, however, use them with the Control file name, the Equate file name, or any Module file names.) Even though you use quotation marks with a "G", "P" or "U" file name, the rule remains that embedded spaces are not permitted.

CHAPTER 14: ARBITRARILY GENERATED LABELS

One of the most significant features of Symbol Master is that it is a symbolic disassembler and generates labels. There are two distinct types: (1) arbitrary labels; and (2) named labels which may, for example, be operating system name labels such as Kernal routines and variables. Arbitrary labels are described in this Chapter. The manner in which lists of named labels for use in a disassembly are created and modified is described in the next Chapter.

The arbitrary labels are based on the hex address of an instruction operand (or .WORD pair), and include the hex notation within the label. However, the hex notation is always prefixed by at least one alphabetic character, so your assembler will consider it to be a valid label. If this label refers to a byte within the range of code and data tables being disassembled, the label will appear in the label field of the referenced line. If you immediately re-assemble, without editing, the hex value embedded within the label will match the address assigned by your assembler. If you use your assembler's editor to insert or delete instructions after a disassembly, or if you reassemble at a different origin, the hex values embedded within the labels will not match the addresses assigned by your assembler. Significantly, however, the assembly will be correct because the assembler will treat the label as a symbolic label, and not as a mere numerical value.

Here is the format of the generated arbitrary labels:

pchhhh

Where --

"p" is the prefix which may be present.

T prefix means the label appears within a data table defined as a block on the editor screen and through which the disassembler has passed.

X prefix means the label is external to the program. In other words, there was no byte corresponding to this label in any of the blocks of code or data tables through which the disassembler passed. This label will also appear in the list of equates at the beginning of the Symbol Master output.

(This is an appropriate point to mention a significant enhancement of Symbol Master V 2.Ø over the V 1.Ø series. V 2.Ø now properly handles self-referenced code (e.g. self-modifying code) where the second or third byte of an instruction is referenced. In this situation Symbol Master puts a label on that line and refers to the referenced byte with a label of the form LABEL+1 or LABEL+2. In addition, a comment is generated to alert you. The label will not be of the external type (will not have an "X" prefix), and thus the list of equates at the beginning of the disassembly will not be cluttered by what really are spurious external references.)

Z prefix means the label refers to a location on page zero. Z supercedes X and T since zero-page locations are nearly always external, and rarely refer to an instruction line. (The main exception is Basic's CHARGET routine in the C64.)

if null prefix, then the label refers to a regular line of instruction code through which the disassembler has passed and has assigned the label to.

"c" is the label code based on its usage as an operand. Label codes are assigned in a hierarchical manner. The highest usage encountered as an operand within the program determines the label code assigned when output occurs, in accordance with the following hierarchy. The list below is ordered from lowest to highest:

M is the lowest. The default if nothing higher applies.

D means data. Is assigned when a memory location is addressed by a non-indexed instruction which does not change the contents. Primarily these are non-indexed LDA, LDX and LDY instructions.

For when the undocumented opcode option is selected (see Chapter 18), we have made some attempt to set up the disassembler's tables so an appropriate label code is assigned for the "extra" op-codes.

V means variable. Is assigned when a memory location is addressed by a non-indexed instruction which changes it. Primarily these are the STA, STX and STY instructions. In addition, the ASL, ROL, LSR, ROR, INC, and DEC instructions when they directly address memory.

Again, for when the undocumented opcode option is selected (Chapter 18), we have attempted to set up the disassembler's tables so appropriate label codes are generated.

For when the 65CØ2 enhanced 65Ø2 instruction set (also Chapter 18) is selected, these additional instructions change the contents of the addressed memory location, and thus result in a "V" label code being assigned: STZ, TSB, TRB, RMB and SMB.

I means indirect. Is asigned when a memory location is used as an indirect address. Examples are instructions having operands of the form (ZP,X) or (ZP),Y. In addition, the operand of JMP (ABS) is so coded.

A means bAse of an indexed address. Examples are operands of the forms ZP,X ABS,X ZP,Y and ABS,Y.

B means target of a Branch instruction.

J means target of an absolute JMP instruction.

In addition, beginning with Symbol Master V 2.Ø, the address defined by a two-byte .WORD pair (see Chapter 11, Section 11.Ø) is treated for this purpose as the operand of a JMP instruction. While the assigned label code will not always be appropriate, often addresses in .WORD pair tables are routine entry points which are otherwise not referenced.

S means target of a JSR instruction.

"hhhh" is the included hex value of the operand, as described in the second paragraph at the start of this Chapter. Zero page addresses have only two places, i.e. "hh", while absolute addresses have four places, i.e. "hhhh".

CHAPTER 15: THE LABEL NAME EDITOR

15.Ø General Concepts and Purpose

Disassemblies are more meaningful when label names can be used, rather than the arbitrarily-generated labels described in the provious Chapter. This Chapter describes the means for setting up files of label names to be used in the disassembly, as well as several label name files we have prepared. The prepared files can be used as is, or you can modify them using the label name editor. Names are limited to a maximium of six characters in length.

The Label Name Editor is a separate program which communicates with Symbol Master itself only through files. We took this approach so a sufficient number of functions could be put in the Label Name Editor, without making the disassembler code itself unduly long.

Separate versions are provided for the C64 and C128, "LABELEDIT64" and "LABELEDIT128", respectively. Each is loaded like a Basic program, and then RUN. The two versions are nearly identical, except for their load addresses ($Ø8Ø1 in the C64 and $1CØ1 in the C128), and are file-compatible. In the C128 there is a Bank Ø version only. At present it only runs in the C128 4Ø-column mode; it is uncertain whether we will convert it to the 8Ø-column mode also. Check the "LIST ME" file for this and other supplemental details.

15.1 Single- and Two-Byte Named Labels

A significant feature of Symbol Master is that it recognizes the difference between single-byte labels (e.g. routine entry points and single-byte labels), on the one hand, and two-byte labels (e.g. two-byte variables and vectors), on the other hand. We term these "S" and "T" respectively, although that is grammatically inconsistent. (Consistent terminology would have been One and Two, or Single and Double, but the characters "O" and "D" would lead to a confusion which "S" and "T" avoid.

When Symbol Master encounters the second byte of a Two-byte label it automatically uses the notation LABEL+1. In the generated list of equates to external label references, the name will appear only once.

Sometimes you will disassemble programs which address only the second byte of a two-byte variable or vector. Symbol Master will do the same thing as just described. If you have the cross-referenced label listing generated at the end of the disassembly, the label "name" will appear in the list, but there will be no references to it. The list will also include "name+1", and identify the address of all referencing instructions.

Note that a related thing happens when Symbol Master recognizes self-referenced code. A label will be assigned for the first byte of the particular line of code, even though the first byte itself is not necessarily referenced. This circumstance is readily apparent from the cross-referenced label listing. Do not overlook that important analysis tool (the cross-referenced listing).

Getting back to the subject at hand, for each label name you put in the list you must assign a hex value. Prefix the hex value with an "S" if it is merely a single-byte name, and prefix it with a "T" if it is the first byte of a two-byte name. Using the "U" command, described below, to read in one of the prepared label name files will give you an example to make this clearer.

15.2 Brief Introduction to the Label Name Editor

When you LOAD and RUN the appropriate editor version, you will see a screen very much like the regular Symbol Master editor screen, but different enough so you will not forget which one you are in. Cursor behavior is the same with reference to the divider line 5. The top line is for inputting commands, and lines 6 through 25 scroll for entering label types ("S" or "T"), hex values, and names.

Some of the Line 1 commands are the same as in the main disassembler editor, but there are some different ones. The Line 1 commands are defined and discussed in Section 15.5 below. But first it is probably more important to describe the syntax of the actual type, hex value, and name entries on lines 6 through 25.

15.3 Syntax of Label Name Entries

Each label entry is of the form:

<t> <hhhh> <name>

Where --

"t" is the type as discussed in section 15.1 above, either "S" for Single byte, or "T" for Two-byte.

"hhhh" is the hex value of the label. All four characters must be entered, even for zero-page labels. A "$" prefix is never used.

"name" is the name you assign to the label. In order to be compatible with all the assemblers, it is limited in length to a maximum of six characters. Thus, the "name" is from one to six characters in length.

15.4 Entering, Deleting and Viewing the Label Names

Internally, the Label Editor maintains the label values as three-byte variables, with "S" or "T" being the most significant byte. The list is maintained in numerical order, regardless of the order you make entries. In this regard, the expression "t hhhh" is handled much like the Basic editor handles line numbers. All the the "S" entries come before any of the "T" entries. Thus "S ØØØ1" comes before "T FFFC".

There is no List function per se to the screen (as there is to the printer as described below), since scrolling accomplishes the same purpose. Moreover, you can cause the scrolling to begin at an intermediate point of a long list by typing on the screen (but not entering with the RETURN key) a type and value just prior to where you wish to start, and then using the CRSR down key to scroll.

To start the entering process, type the first entry in the above symtax, and enter with RETURN. The spaces are optional on when you type (you can run it all together for speed), but the entry will automatically format itself when you hit RETURN.

To avoid typing this every time, you are prompted on the next line with "S" or "T" to match the one you have just done. If you want to change it for the next entry, just do it. Normally you will enter all of your "S"s, then the "T"s, but the order is not important.

To delete an entry, enter its type "t" and hex value "hhhh", but leave the "name" blank. When you hit RETURN, it will be deleted. If an entry is listed on the screen, an easy way to delete is to spacebar over the name, or use the DEL key, and then hit RETURN.

To change an entry, enter the type "t" and hex value "hhhh", as well as the new name. The new name will replace the old.

All the while you are making and possibly deleting entries you will see a status report of the total number of names, and the total bytes. The total number of bytes is important if the Label Name file you are creating is to be loaded into the C64 version of Symbol Master, because there are only about 7500 bytes available for label names and the label table combined. The length of each label name is two bytes, plus the number of ASCII characters in the name. Thus the length is from three to eight bytes. Each entry in the label table which is generated during disassembly is three bytes. Thus, for example, if you type in a label name file which is 4500 bytes long, then 3000 bytes are left for the label table, and the capacity for disassembly will be 1000 labels, which actually would be a rather long disassembly.

15.5 Label Editor Commands

Here are the commands accepted on Label Editor line one. In general these are entered just like those for the main disassembler editor. Several of these are identical to the main disassembler editor commands. Those are indicated by the notation "same" in the list below, and you are referred to Chapter 8 for their description. The remaining ones are described in detail immediately following the list.

```
A -- Check for Address conflicts
B -- Do a Break, usually to monitor.      (Same)
C -- Clear name list
D -- Check for Duplicate names
H -- Hardcopy of List
K -- Cold start the computer              (Same)
M -- Merge (and overwrite)
U -- read in User label name file
W -- Write user label name file
X -- eXit to Basic                        (Same)
@ -- DOS Command                          (Same)
> -- DOS Command                          (Same)
# -- Change current disk device number.   (Same)
```

A -- Check for Address Conflicts

There are no parameters.

This command checks for situations where you have attempted to assign two names to the same hex value. This would confuse the disassembler, and so the Label Editor checks for this situation.

There are two distinct types of conflicts which are checked for.

First, an "S and T Conflict" occurs when you have the same hex value assigned to a name in the "S" list as you do to a name in the "T" list. Since "T" list entries actually refer to two bytes, both bytes of each "T" entry are checked against each "S" entry.

Second, a "T List Conflict" occurs when you have two "T" list entries separated by only one hex value. Thus, the second byte of one is the same as the first byte of the other.

When one of these occurs, you should edit the list accordingly, changing and possibly deleting entries.

The conflict is reported to you on the second screen line, with the hex values involved. They are reported only one at a time, so you will have to repeat the command after you have fixed each problem.

The "A" command is automatically done as a preliminary by the "W" (Write) command. Thus the Label Name Editor will not allow you to write a file defective in this manner.

C -- Clear Name List

There are no parameters.

Clears the name list in memory. Asks if you are sure.

D -- Check for Duplicate Names

Goes through your entire list to see if you have duplicated any names. This is a voluntary command (i.e., the "W" command does not automatically do this), since a duplicate name will not trouble the disassembler. However, it will trouble your assembler when you attempt to reassemble a source file generated by Symbol Master, so use of this command is highly recommended. Again, it is not forced on you.

H -- Hardcopy of List

Syntax:

H [<option>]

This dumps the entire label name file in memory to the printer.

One optional parameter is accepted, the letter "F". If the "F" parameter is used, printer output is formatted with one-inch (six-line) page breaks every 60 lines.

M -- Merge (and Overwrite)

Syntax:

```
M "Ø:<filename>"          Note:  Again, <> enclose the
    or                           required parameters, but are
M Ø:<filename>                   not actually entered.  The
    or                           drive parameter "Ø:" is
MØ:<filename>                    optional, but suggested.
```

Merges a label name file from disk with a list already in memory. This is an important facility which allows you to maintain a library of short label name files on disk, and combine them into a larger file for a particular disassembly.

In case of duplication, the incoming hex values have priority over those in memory, thus the description "overwrite". Refer to the description of changing an entry in section 15.4, above for an analogy. The "M" command works very much as though you are entering on the screen each of the incoming entries. No checking is done. I.e., neither the "A" function nor the "D" function is automatically done.

U -- Read User Label Name File from Disk

Actually, this command could have been identified as "same" in the list above, because it is also one of the commands in the regular editor, and works the same way. It is repeated here because it so closely relates to the label editor.

Syntax:

```
U "Ø:<filename>"
   or
U Ø:<filename>
   or
UØ:<filename>
```

Note: Again, <> enclose the required parameters, but are not actually entered. The drive parameter "Ø:" is optional, but suggested.

The primary use of this command in the Label Name Editor is to read in a previously created label name file for editing, or as the first step in a series of merges.

Unlike "Merge", the "U" command completely replaces the list previously on the screen.

W -- Write User Label Name file to disk

Syntax:

```
W "Ø:<filename>"
   or
W Ø:<filename>
   or
WØ:<filename>
```

Note: Again, <> enclose the required parameters, but are not actually entered. The drive parameter "Ø:" is optional, but suggested.

This command is the complement to the "U" command, and saves the label name file to disk. The file type is USR to ensure it is not confused with a command file.

The "save-with-replace" form of filename is accepted, if you trust it. As an example:

```
W @Ø:MYLABELS.LBL
```

15.6 Provided Label Name Files

We have put on the disk a number of label name files to save you the trouble. You can use these as is, or edit them using the procedures described above. The descriptions here are brief, since you can read in (U command) and list them to your printer (H command) to see what they actually contain.

"KERNAL64.LBL" are most of the C64 operating system names, including zero page, but exclusive of Basic variables.

"BASICZP64.LBL" are Basic zero-page labels for the C64.

"DEFAULT64.LBL" is a combination of the above two.

"BASICROM64.LBL" are the names of selected routines in the C64 Basic interpreter from $AØØØ through $BFFF and $EØØØ through $E422. (Note that the C64 Basic interpreter actually spills over from the so-called Basic ROM into the first part of the so-called Kernal ROM).

"KERNALROM64.LBL" are the names of selected routines in the C64 Kernal ROM from $E453 through $FF48.

"KERNED128.LBL" are entry points and variable names associated with the operating system and editor in the C128.

"RAM1541.LBL", "ROM1541.LBL", and "ALL1541.LBL" are for the 1541 disk drive ROM. See the example in a later Chapter.

15.7 Miscellaneous Label Editor Notes

Here is a bit of miscellaneous information concerning the label editor.

If you exit the Label Name Editor, you can usually re-enter via warm or cold start. To cold start, just enter "RUN". Or from the Monitor, G Ø80D in the C64, or G Ø1C12 in the C128. The warm start addresses are $0810 in the C64 (decimal 2064 for SYS), and $1C15 in the C128 (decimal 7189 for SYS).

The C64 version loads from $0801 to approximately $19FF. A few program variables are at the end, and then the array begins in memory at approximately $1B00. As label names are entered, the array grows upward in memory. Nothing is touched higher than necessary. In the label editor only, each entry has a fixed length of ten bytes. However, the file is written to disk and storred in the disassembler itself in a more compact form, and the "total bytes" reported to you on the screen reflects the actual number of bytes used when the file is read into the disassembler.

You would have to enter over 2400 label names to even reach $8000 in memory, a number unreasonably large, and the Label Name Editor accordingly does not even bother to check for a memory overflow. In the C64, Symbol Master at $8000 and Micromon at $9000 will peacefully coexist with the Label name editor.

The C128 version loads from $1C01, and the array begins at approximately $3000 in RAM Bank Ø. In the memory configuration used, RAM Ø is available up to $BFFF. This is enough for approximately 3600 label names.

Here is a brief description of the file structure in case you want to generate a compatible label name file from another source you may have. Each entry starts with a two bytes which are the value of the label name, in conventional low-byte-first order. Following these two bytes are one to six bytes of the name in Commodore ASCII. The last byte of each ASCII name has bit 7 set as a flag. Thus each entry has from three to nine bytes in total. They are all run together, and the ASCII bytes with bit 7 set are the only means of separating. The "S" list comes first, then the "T" list, with no separator. All entries are maintained in sequential order by value, within each list. The first four bytes of the file are a header. The first two header bytes are the length, in bytes, of the "S" list, and the second two header bytes are the length in bytes of the "T" list. This is the only means of separating the two lists.

The "U" command adds the two length values from the header and compares the result to the actual number of bytes in the file read from the disk. If there is a mismatch, a "header length error" or a "usr file too short error" is reported. Other checks are done on the file at various times, for example to be sure that six ASCII bytes do not go by without one being flagged, in order to ensure the integrity of the data. An editor error will be reported if something is not right. All of this is to protect the disassembler from a bad label name file. If you generate your own, it is strongly suggested that you run it through the label name editor first, before loading it into the disassembler.

A common 6502 programming technique is to use "BIT" instruction to skip the next one or two bytes. If the next one or two bytes actually contain an instruction which is intended to be executed, this instruction will be "hidden" from a simple disassembler. For example, the C64 Kernal has a routine at $F6FB to handle I/O errors. There are nine different entry points, but the only real difference is that the accumulator is to be loaded with a different value (depending on the particular entry point) before the common portion of the routine is entered. Here is that example:

```
F6FB- A9 01     JF6FB LDA #$01     ;TOO MANY FILES
F6FD- 2C              .BY $2C      ;Skip next two bytes
F6FE- A9 02     JF6FE LDA #$02     ;FILE OPEN
F700- 2C              .BY $2C      ;Skip next two bytes
F701- A9 03     JF701 LDA #$03     ;FILE NOT OPEN
F703- 2C              .BY $2C      ;Skip next two bytes
F704- A9 04     JF704 LDA #$04     ;FILE NOT FOUND
F706- 2C              .BY $2C      ;Skip next two bytes
F707- A9 05     JF707 LDA #$05     ;DEVICE NOT PRESENT
F709- 2C              .BY $2C      ;Skip next two bytes
F70A- A9 06     JF70A LDA #$06     ;NOT INPUT FILE
F70C- 2C              .BY $2C      ;Skip next two bytes
F70D- A9 07     JF70D LDA #$07     ;NOT OUTPUT FILE
F70F- 2C              .BY $2C      ;Skip next two bytes
F710- A9 08     JF710 LDA #$08     ;FILE NAME MISSING
F712- 2C              .BY $2C      ;Skip next two bytes
F713- A9 09     JF713 LDA #$09     ;ILLEGAL DEVICE #
F715- 48              PHA          ;SAVE ERROR #
F716- 20 CC FF        JSR CLRCHN   ;CLOSE CHANNELS & SET DEFAULTS
                      (Common code continues)
```

In this example, assuming the routine is entered at $F6FB, the accumulator will be loaded with a value of 1. The processor will consider the next instruction to be BIT $02A9. The BIT instruction does not affect the accumulator, so the next instruction executed will be at $F700, with the accumulator still 1. Execution will drop right through to $F715, with the accumuator unchanged. The same occurs for each of the other entry points.

A simple disassembler will disassemble the block from $F6FB through $F714 as a sequence of eight consecutive BIT instructions.

A symbolic disassembler not equipped to handle this situation would likely do the same, except that its label table would be filled with spurious operand labels, such as one corresponding to $02A9. So not only would the meaning be obscured, but the list of equates to external label references would have spurious labels in it.

Symbol Master effectively handles the above in most situations. The appropriate .BYTE $24 or .BYTE $2C pseudo-op is generated, and the line commented. The next line is disassembled as a one- or two-byte insctuction, depending upon whether the byte value was $24 or $2C. The instruction on the disassembled next line, if a two-byte instruction, can have either an immediate operand or a zero page operand.

Here, in general, are the approach and criteria which Symbol Master uses:

(1) Whenever a $24 or $2C "BIT" opcode is encountered, the next address is checked to see whether it needs a label, i.e., is referenced by another instruction in the program.

(2) If the answer is "no", then the $24 or $2C is considered to be a normal "BIT" instruction, and its operand processed normally.

(3) If the answer in (1) is "yes", then two additional tests are done before concluding a "BIT skip" is intended:

(a) The referencing instruction must be a program flow instruction, i.e., a Branch, JMP or JSR; and

(b) The length of the instruction of the disassembled next line must be correct. Following a $24 byte there must be a one-byte instruction. Following a $2C byte, there must be a two-byte instruction.

If either of these further tests is not satisfied, then the $24 or $2C is considered to be a normal "BIT" instruction, and its operand is processed normally. Likely, though, it is something else, and your analysis by hand will be required.

You will often be alerted to this situation by a "self-referenced" comment in the disassembled code.

You will find a "BIT skip" which Symbol Master does not recognize if you disassemble the included MICROMON. Near the beginning of that code are several BIT instructions used to hold data values (which are actually constants used by the program) in a manner such that the MICROMON "New Locator" instruction can relocate the program. There are program references to the bytes following the BIT instructions, but these references are of the LDA variety, not program flow. The referenced addresses will appear in the list of equates to external label references.

This is simply too complex and rare a situation to have Symbol Master analyze. Any attempt would probably cause more confusion than assistance.

There is one case where Symbol Master's "BIT skip" handling fails entirely, and that is where the "hidden" instruction is reached by some sort of vectored jump. In this situation the address of the "hidden" instruction will not be an operand of another instruction in the program.

Symbol Master V 2.0 optionally handles the undocumented 6502 opcodes, as well as the documented 65C02 enhanced instruction set. (Note these alternatives are quite distinct from each other.) To do this, you must load alternative tables into the disassembler (which are provided on the disk). This must be done with care, as you are deliberately overwriting a part of the Symbol Master code.

17.1 The 6502 Undocumented Op-Codes

We are not here trying to explain this subject in detail, so this will be real brief: There are 256 possible op codes, but only 146 are officially defined as part of the 6502 instruction set. This leaves 110 possible opcodes which are undocumented. Many of these simply cause the computer to crash, but there are others which are useful. Typically they perform two instructions at once. These are not at all guaranteed by the manufacturers of the microprocessors, but others have reported a remarkable consistency from one to the next, and even between the 6502 and the 6510, used in the C64. I have not seen any report on the 8502 used in the C128. Supposedly there are commercial programs which use these which you may wish to disassemble. Also, you could use them in your own programs, although finding an assembler will be a problem.

We have done no investigation whatsoever, and have relied on published investigations of others. The various sources are sometimes inconsistent. We have done our best to resolve the inconsistencies. You are on your own with the results.

Here are our sources: "Transactor", November 1985, Volume 6, Issue 03, pages 50-52, by Jim McLaughlin; "Transactor", March 1986, Volume 6, Issue 05, pages 56-60, separate articles by Raymond Quirling and Noel Nyman; "The Complete Commodore Inner Space Anthology", by Karl J.H. Hildon, published March 1985 by Transactor Publishing, page 22, which in turn refers to "B. Grainger's article in IPUG (Jan. 1981) and 'Programming the PET/CBM' by Raeto Collin West"; and "Program Protection Manual for the C-64 -- Volume II", from CSM Software, pages 71-77B. The address of Transactor is 500 Steeles Avenue, Milton, Ontario L9T 3P7, telephone (416) 876-4741. For CSM Software the address is P.O. Box 563, Crown Point, Indiana 46307, telephone (219) 663-4335.

For mnemonics we used the ones from McLaughlin and Hildon. Where there were slight disagreements, we relied on McLaughlin. They both left out opcode $BB, and for that one we went to Quirling, who assigned a mnemonic LSA

As a very rough quide, here are the "extra" mnemonics we put in, and what they may do. Do not rely on this list; go to the sources above:

ASO - ASL then ORA result with .A. Result left in .A.

RLA - ROL then AND result with .A. Result left in .A.

LSE - LSR then EOR result with .A. Result left in .A.

RRA - ROR then ADC result with .A. Result left in .A and C.

AXS - .A AND .X then store result.

LAX - Do both LDA and LDX.

DCM - DEC memory then SBC result from .A. Final result left in .A.

INS - INC memory then SBC result from .A. Final result left in .A.

ALR - AND .A with immediate, then LSR result. Final result left in .A.

ARR - AND .A with immediate, then ROR result. Final result left in .A.

XAA - AND .X with immediate or memory, result in .A.

OAL - ORA .A with #$EE, AND the result with data, then put result in .A and .X.

SAX - AND .A with .X, then SBC data, result in .X

MKA - AND .A with #$Ø4, result to memory.

MKX - AND .X with #$Ø4, result to memory.

LSA - AND memory with Stack Pointer, result in Stack Pointer, .A and .X.

NOP - Six more NOPs are reported.

SKB - Skip next byte.

SKW - Skip next word (two bytes).

CIM - Crash Immediately.

17.2 Modifying the Disassembler for Undocumented Op-Codes

To cause the disassembler to interpret the undocumented op-codes as instructions you must use the "L" command to load in a table, which will overwrite the regular table. Do a Header (non-relocating) load of the appropriate file (i.e., do not specify a load address); it will load to the right place.

Here are the filenames: For C64 version, use "UNDOC64.EXE". For C128 Bank Ø version, use "UNDOC128/Ø.EXE". For C128 Bank 1 version, use "UNDOC128/1.EXE". Load the C128 tables in the same bank as the disasembler.

17.3 The 65CØ2 Documented Op-codes

There exists a completely different set of documented instructions used in the 65CØ2 processor, a CMOS enhancement of the 65Ø2. Although Commodore computers released to date have not used this processor, you nevertheless may encounter code for it. Since this enhanced instruction is officially documented and guaranteed by the manufacturers, it is not repeated here. One source is the Rockwell "R65ØØ Microcomputer System Programming Manual", January 1983 revision. Also, assembly-language programming books for the Apple IIC will have these instructions.

Here are the filenames for the tables to load in, in the same way as described above in Section 17.2: For C64 version, use "CMOS64.EXE". For C128 version, use "CMOS128/Ø.EXE" or "CMOS128/1.EXE", depending on the bank.

Although not strictly necessary, you will arrive at meaningful disassemblies much faster if you do a bit of preliminary investigation of the program you are studying before running Symbol Master. An exception to this is when you use the Command files we have already prepared as examples. We have modified a machine-language monitor for the C64 (MICROMON) to enhance its power for this specialized purpose.

The enhancements are significant enough that even with C128 program files you may wish to go briefly to C64 mode and use the enhanced MICROMON commands.

In particular, the machine-language load command is modified so that programs can, at your option, be loaded either to their header address (non-relocated load) or to any address you specify (relocated load). This is now the same as the "L" (Load) command in Symbol Master itself. The ending address is reported to you. A completely new "dummy load" routine is added which acts like a load, but doesn't actually do a load. It just counts the bytes, and reports the ending address. This is useful when you don't know for sure how long a program is, and you want to avoid clobbering something else. The final added command is a "V" (View) command which does nothing except read the file header to let you know where the file would load if you were to do a non-relocating (i.e. header) load.

With these tools, you can easily analyze "Autoboot" programs of the type which take over control of your computer immediately when they start to Load in. This even includes copy-protection schemes since they all, no matter how weird their disk formatting, necessarily begin with a boot program which your computer can load in the normal manner. This subject is discussed a bit more in the next Chapter on "Autoboot Programs and Copy Protection".

18.1 The Machine-Language Monitor

The included monitor is one entitled "MICROMON", originally developed by Bill Seiler for the PET computers, and modified many times by others. We obtained the source Code through Brent Anderson, who runs the ATUG Disk Exchange, 200 S. Century, Rantoul, Illinois 61866. He fixed it so the Walk, Break set and Quick trace commands all work on the Commodore 64. This happens also to be essentially the same one which is included on the MAE disk, if you happen to have that assembler.

For convenience, you will find five different copies of the object code on the Symbol Master disk, all identical except for where they load. You should have no problem finding a place to locate one of these in memory. A boot program in Basic is included for easier loading. With each version, the cold-start address is the first address of the program, and the warm start address is +3 beyond that. Each one changes the Basic memory pointers, but only if necessary, to protect itself. I.e., the Basic pointers will never be changed to something higher. For its own variables, the included version of MICROMON uses some locations at the lower end of the stack page (beginning at $Ø14C), ane also some at the end of itself up to the appropriate $xFFF boundary of the 4K memory block it occupies. MICROMON does not use the zero page locations $FB, $FC, $FD and $FE, so there should be no conflict with the many machine language programs which use these. Only disk is supported, not tape. If you need to do a tape load, exit MICROMON with the E command to Basic, then use the Basic LOAD "filename",8,1 command to load, and SYS to cold start MICROMON. MICROMON calls routines in both the Basic and Kernal ROMs.

As a service, we will supply the original, commented source code for the included version of MICROMON on disk for a $10.00 handling charge. Prepaid mail orders only if you are ordering nothing else at that time. Be aware that the source code is in MAE format, so you will have to convert it if you use a different assembler.

18.2 The Added Commands

Described here first are the new commands which are particularly useful in preliminary investigation. Next described is the preliminary investigation using the simple disassembler in MICROMON. Finally, the remaining MICROMON commands are briefly mentioned. They are all pretty much the standard ones. Most of you will already have a machine language monitor and will be familiar with its use.

LOAD FROM DISK

```
.L "0:FILENAME"              -- header load, non-relocating

.L 1000 "0:FILENAME"         -- relocated load
```

The first example loads RAM starting at the address specified by the file header. The ending address (plus 1) is reported to you.

The second example loads to RAM starting at $1000, regardless of the file header. The load address 1000 is in hex, and all four places must be given. The ending address (plus 1) is reported to you.

DUMMY LOAD FROM DISK

```
.(see note) "0:FILENAME"       -- header

.(see note) 1000 "0:FILENAME"  -- relocated
```

The "see note" character is a Shift/L, which produces a Commodore graphics character resembling an "L". Other than that, the syntax is precisely the same as for "L" (Load). A load does not actually occur, but the ending address (plus one) is reported as though it had.

VIEW HEADER

```
.V "0:FILENAME"
```

This command reads the first two bytes of a program file, closes the file, and reports to you in hex the load address represented by that header. This is the address where the file would load on its own if you were to do a non-relocating load.

The VIEW HEADER and DUMMY LOAD commands together allow you to determine in advance where a program file will load, without the risk of clobbering something else. Caution in your conclusions: in many copy-protection schemes, the header address is something totally fictitious, put there in confuse. More on this in the section on "Autoboot Programs and Copy Protection".

18.3 Other Relevant Commands

SIMPLE DISASSEMBLER

```
.D 2000 2005

., 2000 A9 12    LDA #$12
., 2002 9D 00 80 STA $8000,X
., 2005 AA       TAX
```

Disassembles the range specified. The bytes following the address can be modified and entered to change the instruction. Disassembly scrolls up and down under control of the CRSR up/down key.

MEMORY DISPLAY

```
.M 3000 3008

.: 3000 30 31 32 33 34 35 36 37 01234567
.: 3008 38 41 42 43 44 45 46 47 8ABCDEFG
```

Displays memory in the range specified in HEX and ASCII intrepretation. Bytes can be modified and entered. Scrolls up and down

TRANSFER MEMORY

```
.T 4000 4FFF 6000
```

Transfers a copy of the data from memory block $4000 to $4FFF inclusive to the block beginning at $6000. In the example here, the ending address of the copy is $CFFF.

18.4 Preliminary Investigation

The tools described above are the main ones you will need. What you are trying to accomplish is to determine the extent of the program you are disassembling, where the blocks of code are, and where the data tables and reserved variable spaces are, if these happen to be included in the body of the program.

Using the "V", "Shift-L" and finally the "L" commands, get the target program into memory. It eliminates the slight amount of mental arithmitic required if you load it where it is intended to execute, but this is not a requirement. Knowing the starting and ending addresses, go through the program with the MICROMON Simple Disassemble and Memory Display commands to find the code and data tables, and jot down the starting and ending addresses of each to enter with the Symbol Master editor for the first trial run. Often, you will want to refine this a bit after seeing the initial results.

It is usually quite obvious when you are viewing valid code. There are no ???'s in the disassembly, and the sequence of instructions will make sense. There are some programs which frequently switch back and forth between code and data, and these require careful attention. The Memory Display will often show you what is clearly ASCII text.

Be alert also for .WORD tables, and enter these as "WO" or "W-" on the Symbol Master editor.

The transistions between code and data tables require special attention for an optimum disassembly. When going from table to code, it is important to start the code on the correct byte, and if you simply scroll your chances of hitting this are

probably about 50-50. When you have been scrolling down with the Simple Disassembler through tables which disassemble as junk and you suddenly find good code, back up very carefully to find the true start. See whether the first instruction makes sense. Instead of scrolling, start the disassembler at a specific address to catch the correct starting byte.

When you have made the notes, enter the block addresses on the Symbol Master editor. You can either cold or warm start Symbol Master from the Monitor. Also the Symbol Master editor can be loaded from MICROMON and cold started.

Use of Symbol Master is usually an iterative process. The above procedure will get you started. After a few trial disassemblies to screen you can get the disassembly parameters "fine tuned" before disassembling to printer, and finally to disk.

18.5 Remaining Monitor Commands

MICROMON has quite a few other commands. However, they are all the standard ones, and are available from many sources. Rather than waste pages of this booklet with a description of commands not particularly relevant to the purpose at hand, the complete list is merely summarized here:

A	Simple Assemble
B	Break Set
C	Compare Memory
D	Simple Disassemble
E	Exit to Basic with vectors restored
F	Fill memory
G	Go execute
H	Hunt memory
K	Kill. Restore vectors and break.
L	Load RAM from disk
SHIFT/L	Dummy Load
M	Memory display
N	New locator
O	Branch offset calculate
P	Output switcher
Q	Quick trace
R	Register display
S	Save RAM to disk
T	Transfer memory
V	View disk file header address
W	Walk code
X	Exit to basic. Keep Micromon vectors
$	Hex conversion
#	Decimal conversion
%	Binary conversion
"	ASCII conversion
+	Hex addition
-	Hex subtraction
&	Checksum

19.Ø C-64 Autoboot Programs

Here is how to easily deal with autoboot programs in the C-64 so you can disassemble and study them.

If you haven't seen one, you will know it when you see it. Such a program is loaded, in Basic direct mode, via something like the following:

```
LOAD "Ø:BOOT",8,1
```

Suddenly, the program takes over your computer and loads the remaining modules while you watch and do nothing.

There are two usual techniques such programs employ. But first, the effect of the ",1" on the Load command: Without this, Basic always does a relocated load, loading the program to $Ø8Ø1, regardless of the program file header. With the ",1", a non-relocated load is done to the address given by the two bytes which constitute the program file header. Normally, you would be returned to Basic READY mode after a load, but these programs of course do not.

(1) The first technique uses the Basic IMAIN vector at $Ø3Ø2,$Ø3Ø3. Here is a typical example. Using the MICROMON added "V" (View) command, you determine that the load address is $Ø2A7. From the memory map on page 318 of the Programmer's reference quide you see that this is the beginning of an "unused" area. Using the MICROMON added SHIFT/L command, the ending address (plus 1) is reported as Ø3Ø4, meaning the true end is $Ø3Ø3. Assuming you have have MICROMON loaded other than $1ØØØ - $1FFF, leaving this area free for investigation, you now load the boot program into this area, picking an address which keeps the mental arithmetic simple:

```
.L 12A7 "Ø:BOOT"
```

Here the ending address (plus 1) of the actual load is reported as 13Ø4. The program is now in memory at $12A7 through $13Ø3 for study where it will do no harm.

This boot program obviously is intended to load past the end of the "unused" area. Investigation with the monitor reveals code at the beginning of the block. Significantly, however, at the end addresses $13Ø2 and $13Ø3 contain $A7 and $Ø2. These two addresses correspond to intended load addresses $Ø3Ø2 and $Ø3Ø3, the Basic IMAIN vector! Following a Basic Load, Basic jumps through this vector (which normally points to $A483). Now the vector points to $Ø2A7 where the Boot code begins, and takes over. The boot code will include the necessary machine language routines to load in the rest.

(2) The second technique uses the stack. Here is a typical example. Using the MICROMON "V" command you determine that the load address this time is $Ø1ØØ, the beginning of the stack page. Using the MICROMON SHIFT/L command, the ending address (plus 1) is reported as Ø259, meaning the true end is $Ø258. Now load the Boot program into a safe area of memory, where it will be benign:

```
.L 11ØØ "Ø:BOOT"
```

The ending address (plus 1) of the actual load is reported as 1259, so the program is loaded for study at $11ØØ through $1258.

Investigation with the MICROMON M and D commands reveals that almost the entire area $11ØØ through $11FF (corresponding to the intended load area $Ø1ØØ through $Ø1FF) is

filled with $Ø2 bytes. Just above that is code. Here's how it works: The Load routine is of course a subroutine. When the RTS is encountered, it pulls the top address off the stack (believing that's where a subroutine was called from) adds one, and resumes execution at that address. Of course now whatever was at the top of the stack, regardless of the stack pointer, is gone. No matter where in the stack the RTS goes, out will come the address $Ø2Ø2. So, the next instruction executed is the one at $Ø2Ø3, which of course is part of the boot program.

Having recognized either of these approaches (or perhaps another one someone has devised), you apply Symbol Master to disassemble the Boot program. After analyzing how it works, you modify it as necessary to load in the next module under your control.

Often there will be a chain of such modules, where each one loads the next. Just follow it through. WIth Symbol Master you can save all of your work to disk every step of the way.

19.1 Caution on Load Addresses

With the exception of the initial "Boot" program (or any program you yourself load in direct mode), it is not necessarily true that the program file header contains the actual load address. The boot program, or intermediate links, may very well use a relocated load.

From the decsription of the Kernal LOAD routine in the Programmer's Reference Guide, you will see that setting up a secondary address of Ø prior to a Load will result in a relocated load to the starting address specified in the .X and .Y registers, regardless of the header. Watch for this as you go through the file chains.

Incidentally, in the C-64 the Symbol Master "$D" main module is loaded via a relocated losd to $AØØØ, then copied to $DØØØ. The "$A" main file is then loaded to its header address $AØØØ. This time a non-zero secondary address is used for a non-relocating load.

19.2 Copy Protected Programs

Symbol Master is not intended as a "disk duplicator" or a piracy tool. It is for serious study and analysis. If you want to rip off copy protected commercial programs and contribute towards drying up the source and driving up the price, you will save yourself a lot of trouble by buying one of the protection-breaking disk duplicator programs.

Of course you won't have the slightest idea of what is going on, and the contribution to your knowledge as a programmer will be nil.

On the other hand, careful study using Symbol Master as a tool will reward you greatly in increased programming knowledge as you unravel the techniques used in these programs. Within limits, the more complex the task, the more you will learn. Symbol Master minimizes the drudgery, leaving you free to apply the thought.

Also, if you have some favorite programs which are protected by the bump and grind method whereby your 1541 disk drive is destroyed, your efforts can be rewarded with an executable version of the program which loads and executes in the normal manner. Please, please do this for yourself, not for your friends.

Anyway, if you followed the discussion above concerning "Boot" programs in general, you already know how to deal with a copy protected program. There is no difference!

Here's why: Every disk, no matter how heavily "protected", must have a boot program which loads in the normal manner. Otherwise it could not start. Once you look at the boot program, you might find anything. But you will be able to see it.

Some simple protection methods (e.g. the bump and grind method) have deliberate bad sectors on the disk so the DOS issues a read error. The boot program (or the next link) will look for that error, and abort if it is not found. All you have to do is find that portion of the code and reverse the logic so it runs if there is no error. Others, i.e. those which modify DOS will require more study. But remember, if they can do it, so can a modified program you write.

You may also encounter encrypted program files. With these, even if you are able to do a load from the disk, they are utter nonsense. However, the boot program, or intermediate program, will necessarily have the decryption algorithm. Once you find the decryption code, just reassemble it in a program you control.

19.3 Mangled Disk File Names

Most copy protected disks have disk directories which are odd, to say the least. One technique for doing this is to include non-ASCII characters in the filename. The simplest example is a zero byte. The DOS has no trouble finding these files to load; it just can't put the directory listing together.

If this technique has been employed, you will discover the filenames to use as you go through the chain beginning with the "Boot" program. Another way is to use a disk utility which allows you to examine blocks on the disk. You can see how the directory is set up. You don't need anything fancy to do this. The "DISPLAY T & S" program on the Commodore "Test Demo" disk will do quite nicely, provided you keep track of when you have a decimal number and when you have a hex number. "DISPLAY T & S" is a good program to use because it does not have modify capability, and therefore will not let you damage a disk.

19.4 File Reading Program

We have included a little utility which is a modified version of Butterfield's "COPY FILE 64" program. The original version prompted you to input the file name, which was then read from one disk and written to another disk. That is fine for ordinary files, but can't handle a scrambled file name. The original version consisted of a short Basic program with a machine language program appended. We pulled off the machine language program, and reassembled it to load and run at $C000. The Basic portion can now be edited each time you use it, so long as you don't make it much longer.

Here's how it works. First, load the machine language portion from Basic direct mode using:

```
LOAD "0:CPYFL64/MOD.EXE",8,1
```

Next enter NEW followed by CLR to reset the Basic pointers. Then:

```
LOAD "0:CPYFL64/MOD.BAS",8
```

LIST the Basic program so you can edit it. Edit line 20 so that X$ is the filename, strange bytes and all, as you have determined it. For example, if you have determined that the filename consists of a zero byte followed by ASCII "MD2", here's what to do:

```
20 X$ = CHR$(0)+"MD2"
```

With the CHR$ function and the string concatenation operator you can achieve anything required. Keep in mind that the disk directory or boot program will give you the strange bytes in hex, and you must convert these to decimal for the CHR$ function.

In line 30, assign an ordinary filename to Y$.

Insert the "from" disk in your drive and run the program. The file will be loaded. When the prompt comes, put in another disk, and a duplicate file will be created, which you can then thereafter load with MICROMON or Symbol Master.

19.5 C-128 Boot Programs

It is safe to assume that most commercial programs for the C128, whether copy-protected or not, will take advantage of the "Boot Sector" on the disk, which is Side 1, Track 01, Sector 00. Assuming the disk drive is present, on reset the C128 "Boot" routine reads this sector and loads it into the cassette buffer $0B00 through $0BFF in RAM Bank 0. The first three bytes of the block are checked to see if they contain the ASCII key "CBM". If not present, the boot is aborted, and initialization proceeds, leading to the Basic opening message.

What happens next if the "CBM" key is present is described in other sources. It is dealt with in the Abacus book "128 Internals", and the magazines are beginning to treat this subject. You can learn more even by studying the boot sector creator program which comes with the 1571 disk drive.

In any event, while the rules are more complex, you can trace through what will happen, and thus begin the disassembly process.

CHAPTER 20: SPECIFIC ASSEMBLER NOTES

In this section is information specific to each of the assemblers supported. This section includes the description of the specific effect of the "OPT" option which may be specified on the miscellaneous parameter line (editor screen line 3). You will want to read the portion pertaining to your assembler in particular, and perhaps briefly consider the others.

20.1 MAE Assembler

Effect of "OPT" on the miscellaneous parameter line -- OPT selects the explicit zero page addressing mode. While it is not necessary with current versions of the MAE assembler, the assembler still supports it. If you like to see the asterisk (*) before the operand on zero page addresses, Symbol Master will give it to you.

Not selecting the OPTion will not cause a phasing error on assembly because Symbol Master always outputs the equates before any program lines. Thus all zero-page references are defined before they are referenced, and MAE will be content.

MAE does not (to my knowledge) support explicit absolute address addressing mode. There are times when a programmer deliberately wants the absolute addressing mode, even though the address involved is on zero page. Typically the reason is to prevent indexed addressing from wrapping around to the start of zero page, when the intent is to continue on to page 1. A slight problem comes when re-assembling source code generated by Symbol Master: Since the address is a zero page address, MAE will assign the zero page addressing mode. This will make the program a byte short, and it may not even run if the wrap around effect referred to above is a problem.

Fortunately, Symbol Master flags this situation for you, and you can easily fix it. The situation will be flagged by Symbol Master commenting the line ";ABS. MODE FOR Z.P. ADDRESS". When you see this, use the .BY pseudo-op on one line to generate the correct absolute mode op-code in hex, and then use the .SE pseudo-op on the next line to reference the operand by its asigned label. After you do this, you can do an assembly, and recreate the original exactly.

Each file generated by Symbol Master has the header properly written to tell MAE the precise length in bytes of the file. No manipulation on your part is required to accomplish this. Symbol Master itself imposes no limits on the MAE file length. However, to avoid an !0F OVERFLOW IN TEXT FILE error message issued by MAE, take care not to disassemble too large a chunk of code as one file. Once a program has grown to the size where it will not all fit in the MAE text buffer at once and a Control File with .CT and .FI pseudo-ops must be set up, there is essentially no speed advantage in trying to make the files as long as possible.

Symbol Master assigns MAE line numbers beginning with line 1, and uses an increment of 1. You will probably want to change the increment to 10 using the MAE]NUMBER command the first time you edit the file. Symbol Master does not assign an increment of 10 in order to minimize the chance of overflowing the numbers. A module length of 1000 lines is not impossible, which is all an increment of 10 would allow.

20.2 PAL64 Assembler

Effect of "OPT" on the miscellaneous parameter line -- OPT controls the line numbering when writing multiple module files to disk. In the default mode (OPT not specified), the line numbering is restarted with number 10 with each new module file written to disk. When OPT is specified, line numbering continues in sequence even though a new file is opened.

In either case, the line number increment is 10. This allows you to conveniently insert lines using the the Basic screen editor even if you do not have the separate "POWER 64" utility program from Pro-Line Software Ltd., or a similiar Basic aid program.

PAL supports explicit (forced) absolute addressing mode using an exclamation mark (!) as a prefix to the operand. When PAL mode is selected, Symbol Master generates the "!" where appropriate, and also adds a comment to flag for you what is an unusual situation.

In virtually all cases, Symbol Master disk output can be immediately reassembled perfectly. We have encountered one exception with PAL. That exception is where a colon (:) occurs in the comment field when interpreting BYTE table data as ASCII. Even though this is in what other assemblers would consider the comment field, PAL thinks a new line is intended. It is unlikely what follows the ":" is a valid instruction, and a Syntax error, likely non-fatal, ordinarily results. The solution is to edit out the ":". We weighed the pros and cons of whether Symbol Master should consider ":" to be a prohibited character when interperting ASCII in the syntax of the PAL assembler. Obviously we determined to leave it in. If this does happen to you, you will now know why.

The PAL64 assembler can be co-resident in memory with Symbol Master. However, beware and be aware that PAL is longer than 4K. If does not start at $9000 but, rather, a bit lower. So do not have PAL loaded at its normal place and think you can load the Symbol Master editor or MICROMON at $8000 through $8FFF. If you try this, you will clobber PAL. The remedy is to load PAL last. PAL will look at the Basic memory pointers advusted by Symbol Master or MICROMON, and load itself below that.

20.3 Develop-64

Effect of "OPT" on the miscellaneous parameter line -- OPT selects verson source file output compatible with earlier versions of Develop-64, i.e., version 3.0 series. The default of no OPTion causes source files compatible only with version 4.0 and above to be generated. The Develop-64 version 4.0 documentation explains that source files created with the earlier versions are compatible with version 4.0, except they are slightly longer, and except that lines with the BYT pseudo-op must be reprocessed by the editor. The specifics are as follows: Develop-64 version 3.0 used the Basic INPUT instruction to read lines of source code from tape or disk. A leading quotation mark (") was necessary to prevent INPUT from stopping at the first comma (,). Develop-64 version 4.0 uses a machine-language routine to read the source code lines. The quotation mark is not necessary, and the elimination of it makes the lines shorter. Regarding the BYT, version 4.0 encodes these slightly differently. The reason is theirs to know.

Develop-64 version 3.0 limits the maximum number of source code lines to about 1000. So don't try to dissassemble too long a program if you are generating a source code file for version 3.0. Symbol Master does not keep count for you. Version 4.0 allows the use of the LIB pseudo-op to refer to as many module files as desired and will fit on a single disk, so

there is no problem in this regard when disassembling for Develop-64 version 4.Ø. Also, do not specify the Multiple file mode for Develop-64 version 3.Ø. Symbol Master will go ahead and do it, but version 3.Ø will be unable to read in the multiple files since it doesn't support the LIB pseudo-op.

Filenames in Develop-64 -- As you know, when reading or writing source code files Develop-64 automatically adds the extension ".SRC" to the filename. In order to avoid the total filename length exceeding sixteen characters, the unique portion you specify cannot exceed twelve characters. In Develop-64 mode, Symbol Master does the same thing. Thus when creating your filenames, limit the length to twelve. Symbol Master will then add the ".SRC" extension. If your name is too long, Symbol Master will truncate as necessary to fit. However, in order to allow the filename form "Ø:filename" used with dual disk drives (and advisable even with a single disk drive), Symbol Master bases its truncation on a filename length of eighteen characters. To conclude: Limit filenames to 12 characters. As a matter of caution, examine the disk directory after writing files for Develop-64. If the ".SRC" is not properly appended (it may be cut off), use the DOS Rename command (@RØ:newfilename=Ø:oldfilename) to remedy the situation. If a control file is involved, you may also have to edit the filename following the LIB pseudo-op.

2Ø.4 Commodore Assembler (ASSEMBLER64)

Effect of "OPT" on the miscellaneous parameter line -- none. The OPT will be accepted by the editor, but at present has no effect on the disassembly.

The lack of an explicit zero page addressing mode can cause the same situation as discussed above for MAE. Refer to that discussion for the remedy which will exactly recreate the original object code upon reassembly. The pseudo-ops are, however, different. Use .BYTE on one line to generate the correct absolute mode op-code in hex. and then the .WORD pseudo-op on the next line to reference the operand by its assigned label.

2Ø.5 LADS Assembler

Effect of "OPT" on the miscellaneous parameter line -- exactly the same as for PAL. Refer to the first two paragraphs of the PAL description.

LADS is the only of the assemblers supported which does not have a .WORD pseudo-op. Therefore "WO" and "W-" blocks (see Chapter 11, Section 11.Ø) do not generate addresses, and the analysis is not as good.

LADS does not have an explicit zero-page addressing mode, but since it doesn't have .WORD either the fix if you get a message "ABS MODE FOR Z.P. ADDRESS" will be more complex. Probably careful use of the .BYTE pseudo-op can take care of it. Refer to the MAE description, 2nd and 3rd paragraphs for more discussion.

A standard (unmodified) LADS assembler requires the operands of .BYTE pseudo-ops to be in decimal, so Symbol Master gives them this way for LADS. Also, binary immediate operands are not recognized, so Symbol Master uses hex operands for AND, ORA and EOR immediate in LADS mode.

Since you can modify the LADS assembler, if you like and are determined to use LADS, we suggest modifying it to accept PAL source code so your disassemblies and reassemblies will be more meaningful.

20.6 Merlin Assembler

Effect of "OPT" on the miscellaneous parameter line -- none. OPT is accepted by the editor, but has no effect on the disassembly in Merlin (MER) mode.

The most significant thing to note is that Symbol Master in Merlin mode writes what Merlin calls "text" files to be read in by the Merlin "R" command, rather than "source" files to be read in by the Merlin "L" command. Be sure to enter NEW to clear out the Merlin editor before doing the Merlin "R".

The reason Symbol Master writes Merlin "text" files is that the Merlin "PUT" pseudo-op for "inserting" a text file in an assembly only works with "text" files. It seemed strange to us to have Symbol Master write one type of file when doing an "S" (Single) file disassembly to disk, and another type of file when doing an "M" (Multiple) file disassembly to disk, so we selected the form which is usable with either. Of course if you prefer Merlin "source" files, you can read the "text" files into the Merlin editor, and then Save them back to disk in converted form.

Merlin supports explicit (forced) absolute addressing mode using a colon (:) as a suffix to the mnemonic. When Merlin mode is selected, Symbol Master generates the ":" where appropriate, and also adds a comment to flag for you what is an unusual situation. In virtually all cases, Symbol Master disk output can be reassembled perfectly.

20.7 Panther Assembler

Effect of "OPT" on the miscellaneous parameter line -- none. OPT is accepted by the editor, but has no effect on the disassembly in Panther (PAN) mode.

The most significant thing to note is that, in Panther mode, Symbol Master writes untokenized "ASCII text files" to disk, rather than tokenized source code files. Thus you must use the Panther READ command to get them into the assembler's editor. The Syntax, however, will be correct, and a single file can be immediately reassembled.

A slightly cumbersome aspect to this is that the Panther LOA pseudo-op for including multiple files in an assembly requires a tokenized file saved with the SAVE command. Thus, if you have done a multiple file disassembly to disk, you will have to process the Equate file, as well as each Module file through the Panther editor to convert it from "ASCII text file" to "tokenized source code file" form.

The lack of an explicit zero page addressing mode can cause the same situation as discussed above for MAE. Refer to that discussion for the remedy which will exactly recreate the original object code upon reassembly. The pseudo-ops are, however, different. Instead of .BY (equivalent to the standard .BYTE), use DFC on one line to generate the correct absolute mode op-code in hex. Instead of .SE (equivalent to the standard .WORD), use the ADR pseudo-op on the next line to reference the operand by its assigned label.

Using command (.CMD) files, we have provided you with a number of examples which, perhaps, will be more helpful than this documentation. We have tried to illustrate a number of situations.

21.1 Example 1

Example 1 is the DOS wedge disassembly using the C64 version covered in the "Quick Start" procedure at the beginning of this manual, Section 1.1.

Here's a brief analysis of the results. At $CC03 and $CC0E are two tables, which contain the high bytes and low bytes, respectively, of the DOS command action addresses. Actually, you need to add +1 to each address to see where they point to because the action routines are reached via an RTS after the address is retrieved from the table and pushed on the stack at $CD3F - $CD47. RTS always adds +1 to the address on the top of the stack. The ASCII table at $CC19 contains the commands the wedge recognizes. You may see some commands you didn't realize the wedge has. Of course the positions in all three tables correspond. If you reassemble, you will probably want to assign names to the commands; put NAME-1 in the table, and use the appropriate operator for your assembler to specify the high byte for the first table, and repeat with low bytes for the second. Using that technique the program can be reassembled at another address, and instructions can be added and deleted.

Beginning at $CC24 has the look of a data area. Some is probably unused. Beginning at $CC7B is an ASCII data table with the start-up message. At $CCDE is a JMP instruction with the asigned label "ACCDE" indicating data, not program flow! The explanation is that this is in fact data, as the next five instructions in the program copy this over the top of the Basic CHRGET routine.

21.2 Example 2

Example 2 is the same example, but in the C128 mode, as described in Section 1.2. The point here is that programs for one computer can be loaded into another for analysis.

21.3 Example 3

Example 3 is the same DOS wedge, only loaded to a place in memory other than where it runs. Use the C64 version and, from the Symbol Master editor, Get the command file "DOS5.1$3.CMD". Then do a relocated load:

```
L 3C00 "DOS 5.1"
```

If you prefer, the file could be loaded from MICROMON with the same command, except following the period (.) prompt.

21.4 Example 4

Example 4 is the 8K Commodore 64 Basic ROM. For the ROM examples we have set up, keep memory free from $2000 through $5FFF. Load the Symbol Master editor and MICROMON at other locations. There are many 4K blocks available. Use "U" to load the Label file "DEFAULT64.LBL"

First, from MICROMON, make a RAM copy of the Basic ROM:

```
.T A000 BFFF 2000
```

This will create a copy of the Basic ROM from $2000 through $3FFF. This step is necessary because Symbol Master runs with the Basic ROM banked out. If you tell Symbol to look in the range $A000 through $BFFF it will not find Basic there.

The prepared command file to Get then Run is "BASIC.CMD".

It is possible you will want to generate multiple source files to disk for your assembler to recreate the Basic ROM. If you are not accustomed to large assemblies, this may take a moment to set up. With MAE you will need to lower the bottom of the label buffer to $1000. The default will overflow. Likewise, Develop-64 will overflow its symbol table unless you change the configuration. We have used the two areas $4800 - $5FFF and $6000 - $9FFF as the combined symbol table. The maximum of 800 source statements in this configuration is not a limitation since the assembly will proceed from the disk file. No adjustments are needed to PAL or CBM. In all cases, start the re-assembly with the control file.

Symbol Master's source code output is of course not commented, except for occasional information. If you want a commented version, we highly recommend "What's Really Inside the Commodore 64", by Milton Bathurst, which we also sell under the brief identification "C64 Source". Nearly every line is commented, and every routine is introduced with a statement of its purpose. Both the Basic and Kernal ROMs are covered.

21.5 Example 5

Example 5 is the Commodore 64 Kernal ROM. There are actually four different versions of the Commodore 64 Kernal ROM, counting the SX-64 as one of these. We have prepared a command file for each. Determine which ROM is in your machine, and use the appropriate command file.

The byte at $FF80 (decimal 65408) identifies the ROM version. Either PRINT PEEK this location from Basic, or use MICROMON to examine it in hex.

	Hex Value	Decimal Value	Command file
ROM1	$AA	170	"KROM1.CMD"
ROM2	$00	0	"KROM2.CMD"
ROM3	$03	3	"KROM3.CMD"
SX64	$60	96	"KROMSX.CMD"

Use MICROMON to copy your Kernal ROM to RAM:

```
.T E000 FFFF 4000
```

The procedure is the same as for the Basic ROM, Example 4 above.

21.6 Example 6

This example is the C64 version of Symbol Master itself. Load the Editor at $C000. Get the command file "SYMBOL.CMD". Use the "U" command to load in the label name file SYMBOL64.LBL. Then Run the disassembler, using the R command. Symbol Master can disassemble itself from under the Basic and Kernal ROMs because both ROMs are banked out while Symbol Master does its work. This disassembly will well demonstrate that different modules of a program need not be contiguous in memory for a proper disassembly. This example is set up for the editor loaded at $C000.

21.7 Example 7

This example is the C128 operating system ROM. Load the Bank Ø version of Symbol Master. Use G to Get the command file "KERNED128.CMD". Use U to load the label name file "KERNED128.CMD". Run the disassembler. Have patience, this is a long disassembly.

21.8 Example 8

This is the ROM of the 1541 disk drive. Use the C64 version of Symbol Master.

Exit to Basic and LOAD and RUN the program file "COPY/DISKROM.BAS". This program pulls a copy of the 16K ROM out of the drive and puts it into the C64 at $4ØØØ through $7FFF. This takes about 45 seconds, since 16K must be transfered over the serial bus. For purposes of information, the file "COPY/DISKROM.PAL" is the PAL source code which generated the ".BAS" file. Even if you don't have a PAL assembler you can list it to see how it works.

Use the command file "1541.CMD", and the label name file "RAM1541.LBL". Run the disassembler.

There have been many versions of the 1541 ROM, so you may have to adjust the command file to match yours.

Superb comments for this disassembly can be found in the book "Inside Commodore DOS" by Dr, Richard Immers and Dr. Gerald G. Neufeld, published by DATAMOST< Inc. and Reston Publishing Company.

The program "COPY/DISKROM.BAS" will also read out the ROM from an MSD drive. For this disassembly, load the table for the 65CØ2 instruction set as described in Chapter 17.

22.0 Further Notes

Find and list the "LIST ME" file on your disk for updates to this manual, and any other information pertaining to the version of Symbol Master supplied on your disk.

22.1 C128 Capabilities

I am not certain whether the C128 editor will be working fully, but it is quite useable in any event. In particular, 80-column scrolling may not be operational. If so, you cannot create command files longer than 20 blocks (although longer ones can be read in). The 40-column editor for Bank 0 works as of this writing, but not the Bank 1 editor, so the same limitation may apply. Refer to the "LIST ME" file for details.

22.2 C128 80-Column "Fast" Mode

This works. Turn on the computer, and get the 80-column display going. Enter the Basic "FAST" command. Then Load Symbol Master. (Again, you may be limited in screen editing capabilities to 20 lines, but the disassembler will in fact run twice as fast in this mode, which is nice for long disassemblies, particular for printing cross-referenced label listings.

One caution, the C128 Editor ESC/X sequence will work to toggle output between the 40 and 80 column screens, and you may wish to use this to get the 40-column editor. However, it is important to always exit Symbol Master in the same screen mode you entered. Otherwise the Kernal Editor variables get all messed up when Symbol Master swaps out zero page.

You can easily go between the "fast" and "slow" modes from the C128 Monitor by calling the routines in the Basic ROM. For "fast", do J F77B6. For "slow", do J F77C7. Here is a complete procedure to follow in case the 80-column editors are not working: Turn on the C128 in 80-column mode. While still "slow", Load Symbol Master. In Symbol Master, use ESC/X to go to the 40 column screen to edit. When done, ESC/X again back to 80 columns. Break to the monitor, and do J F77B6 for "fast". Then "G" to warm start Symbol Master. (It's easier to do than to describe.)

22.3 Error Conditions

(1) The label table has a capacity of 2500 labels in the C64 version. In the unlikely event this is exceeded, a label table overflow message will be issued. The label value where the overflow occured will be printed, so you can tell how far the disassembly progressed. Execution aborts, with warm start entry.

(2) Symbol Master creates a label table, refines it, and refers to it often on subsequent passes. If, on a subsequent pass, a label is missing which should logically be in the table, an execution error is detected, a message is printed, with abort and warm start. The error parameters printed are: the address in Symbol Master where the execution error occurred, the value of the missing label, the program counter with reference to the offset origin, and the program counter with reference to the intended origin of the program being disassembled.

In general this error will occur anytime the contents of memory through which Symbol Master is being sent changes from pass to pass, particularly if a "Code" module is involved. If you disassemble through memory where Symbol Master keeps variables, for example, you will get this error. Also, disassembling through screen memory when output is occurring will cause this. Going though the I/O register area in the C128 also does this. In the C128 there are quite a few logical banks where there is not memory actually installed, for example, where "internal" or "external" "function ROM" is specified. These areas read out random junk, and cause an execution error.

(3) Symbol Master has no numbered error codes. A text message is always printed. However, you may get an I/O error message generated by the Kernal. I/O error #4 means File Not Found. I.O error #5 means Device Not Present.

22.4 Defining Blocks on the Editor Screen

The following was written before the V 1.Ø manual before the editor "Fix" commands were put in. The following is much less of a problem now, but still deserves mention:

(1) Proofread the blocks very carefully. The Hardcopy dump helps. You will have three columns defining the starting, ending, and intended origin address of each block. Assuming they are intended to be contiguous, run down the list columns one and two to be sure you stay in sequence from one block to the next. Compare columns one and three of each block to be sure they always have a fixed relationship.

(2) A good sign that you have made an error of the type just described in (1) is when a ";NEW ORIGIN" comment is issued where you didn't intend one.

(3) If you inadvertently overlap the definitions of memory blocks you are sending Symbol Master through, duplicate labels will be generated, in addition to the ";NEW ORIGIN" comment. Your assembler will choke on this.

22.5 Printing of Messages

(1) All execution messages are printed to the screen, followed eventually by the warm start message. You are always reminded where you have the editor loaded. When you press any key, the screen is cleared and you are back to the editor screen.

(2) Just prior to opening a disk file during execution, the filename is printed to the screen. If a DOS error occurs (e.g. "File Exists"), that will be printed next, and you will know exactly what happened.

(3) Where possible, editor messages are printed to the second line, which has no other function. Some (e.g. the Kernal LOADING message) cover more. When this occurs you will need to re-initialize the editor, for example by hitting RETURN on a blank line.

22.6 Halting and Pausing Execution

(1) On output to Screen you can slow down the output using the CNTRL key. You can pause it with the STOP key. RETURN resumes. If you want to study the screen display a few lines at a time during execution, you can alternate back between STOP (PAUSE) and RETURN as often as you like.

(2) While the Screen is Paused by action of the STOP key, the DEL key will abort, and return to warm start. The DEL key has this effect only during Screen pause.

(3) On output to Printer or Disk, the STOP key will abort. There is no pause function. Particularly with the printer, you may have to work the STOP key a few times, or hold it, because it isn't scanned very often while the printer has the serial bus tied up.

(4) The Disk Directory display accessed via the editor DOS manager is a bit different. In that case, SPACEBAR pauses and resumes. STOP stops entirely and closes the disk channel. RETURN returns to the editor screen when finished.

(5) In the C128 the "NO SCROLL" toggle key operates during output to the screen. However, to abort a disassembly you will still have to use the STOP and DEL keys.

22.7 Random Tips

(1) In the C64, if a program file is normally loaded from Basic with the command LOAD "filename",8 (with no ,1), then the intended load address is $0801, regardless of the header. Sometimes the header will be $0400, but don't be confused by that. It just means the file was originally saved on an older Commodore PET machine.

(2) If a byte you have preliminarily identified as belonging to a table because it does not appear to be code ends up geting assigned a label of the form "TJhhhh", "TShhhh" or "TBhhhh", and it is not being referenced by a .WORD byte pair, something is not as it seems because JMPs, JSRs and Branches should not be going to junk tables. You should strongly suspect that a portion of the target program has been encrypted. A less likely possibility is use of the undocumented op-codes.

(3) If a program is loaded from Basic, and you know it has a SYS to a machine language routine but you can't list it, suspect a compiled program or one with shifted characters in REM statements. Some programs also end a Basic program line with a string of ASCII DEL characters ($14), which backspace over the line before you have time to read it from the screen. Careful examination with MICROMON is in order.

22.8 Disassembling Cartridge ROMS in the C64

You may wish to disassemble a cartridge ROM in the C64 at $8000 thru $9FFF. A slight complication is that Symbol Master in the C64 runs in a configuration with all 64K RAM, I/O out (and IRQs off). Thus, if you tell Symbol Master to disassemble from $8000 through $9FFF, it will find RAM, likely filled with junk, rather than the cartridge.

The solution is the use the MICROMON Transfer (.T) command to copy the cartridge ROM to another area of RAM. This is the same technique described in the examples of Chapter 21 for disassembling the Basic and Kernal ROMS. You could also write a little routine to copy the cartridge ROM to the underlying RAM at the same address.

Brought to you by:

https://www.facebook.com/groups/commodoreinternationalhistoricalsociety

commodore international historical society