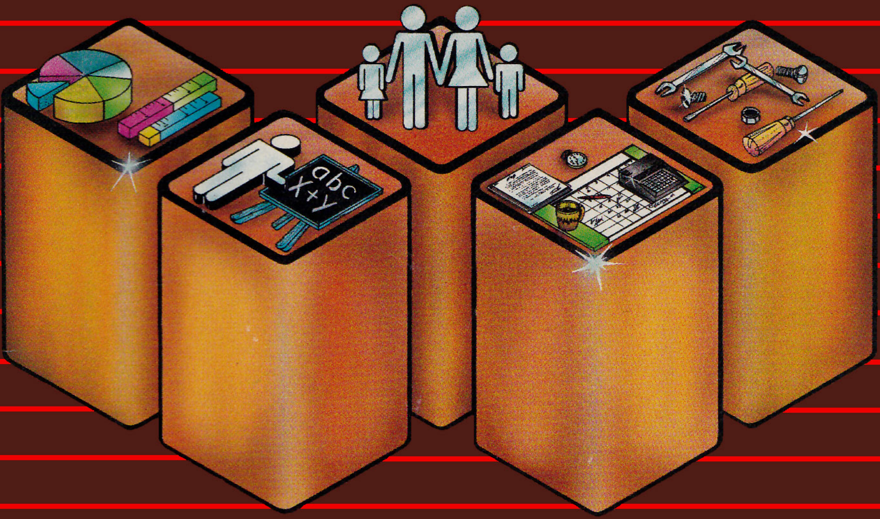
MERLIN PRO™

The Macro Assembler
For The Apple IIe & IIc

By Glen Bredon



RogerWagner™
PUBLISHING, INC.

MERLIN™ PRO

The Macro Assembler For The Apple IIe & IIc

By Glen Bredon

INSTRUCTION MANUAL

ISBN 0-927796-04-X

PRODUCED BY:

Roger Wagner™
PUBLISHING, INC.

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

OUR GUARANTEE

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

PRODUCT REFERENCE:MERPRO 2M0485LC

First, the stuff our lawyers make us say . . .

ROGER WAGNER PUBLISHING, INC.
CUSTOMER LICENSE AGREEMENT

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

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

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

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

LIMITATION ON WARRANTIES AND LIABILITY

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

Then Apple´s...

APPLE COMPUTER, INC. MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, REGARDING THE ENCLOSED SOFTWARE PAKAGE, ITS MERCHANTABILITY OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

And now, on with our program!

ABOUT THE AUTHOR

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

MERLIN PRO

Merlin Pro is an extremely powerful, comprehensive Macro Assembler system for the Apple //e or Apple //c computers. It consists of four main modules and numerous auxiliary and utility programs which comprise one of the most complete assembler systems available for ANY personal computer! Merlin's four main modules are:

- EXECUTIVE system, for disk I/O, file management, ProDOS interpreter, etc.,
- EDITOR system, for writing and editing programs with word-processor-like power,
- ASSEMBLER system, with such advanced features as Macros, Macro libraries, conditional assembly, linked files, etc.,
- LINKER system, for generating relocatable code modules, library routines, run-time packages, etc.

But Merlin Pro is more than just the sum of these four parts. Here are some of the other features offered by Merlin Pro:

- Assembles programs written for the 6502, 65C02 and 65802 microprocessors. (The 6502 for the Apple //e, the 65C02 for the Apple //c, and the 65802 for .... who knows?),
- Merlin Pro comes with TWO assemblers, one for each Apple // operating system: DOS 3.3 and ProDOS,
- Merlin Pro recognizes over 50 Pseudo Opcodes for extreme programming flexibility,
- Merlin Pro has over 40 editing commands for ultimate editing power equaled only by word processors,
- Merlin Pro comes with a complete, commented, disassembled source listing of Applesoft BASIC,
- Merlin Pro comes with a powerful symbolic disassembler to generate Merlin source code from raw binary programs,
- Merlin Pro comes with many sample programs, libraries and other aids to get you going with assembly language fast,
- Merlin Pro is UNLOCKED and COPYABLE for your benefit!

I. INTRODUCTION

II. BEGINNERS GUIDE TO USING MERLIN

III. EXECUTIVE MODE

IV. THE EDITOR

INTRODUCTION

Assembly Language Whys and Wherefores

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

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

From the ground up, your computer understands only two things, on and off. All of its calculations are handled as addition or subtraction, but at tremendously high speeds. The only number system it comprehends is Base 2 (the Binary System) where a 1, for example, is represented by 00000001 and a 2 is represented by 00000010.

The 6502 microprocessor has five 8-bit registers and one 16 bit register. All data is ultimately handled through these registers by a machine language program. But even this lowest of low-level code requires a program to function correctly. This "program" is hard wired within the 6502 itself. The microprocessor program functions in three cycles: It fetches an instruction from computer memory, decodes it and executes it.

These instructions exist in memory as one-, two- or three-byte groups. A byte contains 8 binary bits of data and is usually notated in hexadecimal (base 16) form. Some early microcomputers allowed data entry only through 8 front panel switches, each of which when set on or off would combine to produce one binary byte. This required an additional program in the computer to monitor the switches and store the byte in memory so that the 6502 could interpret it.

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

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

Before using this or any other assembler, the user is expected to be somewhat familiar with the 6502 architecture, modes of addressing, &c. This manual is not intended to teach Assembly Language programming. Many good books on 6502 Assembly programming are available at your local dealer; some are referenced here.

SYSTEM REQUIREMENTS

* APPLE //C or
* 128K APPLE //E with EXTENDED 80 COLUMN BOARD
* VIDEX ULTRATERM (optional)

Suggested Reading:

SYSTEM MONITOR - Apple Computer, Inc. Peeking at Call-Apple, Vol I.

APPLE II MINI-ASSEMBLER - Apple Computer Inc. Peeking at Call-Apple Synertek Programming Manual., Synertek 6500-20.

PROGRAMMING THE 6502 Rodnay Zaks, Sybex C-202.

THE APPLE MONITORS PEELED - WM. E. Dougherty, Apple Computer, Inc.

A HEX ON THEE - Val J.. Golding, Peeking at Call-Apple, Vol. II.

APPLE II REFERENCE MANUAL - Apple Computer, Inc.

EVERYONE´S GUIDE TO ASSEMBLY LANGUAGE - by Jock Root
A continuing series of tutorial articles in SOFTALK magazine. An excellent introduction, easy-to-follow for the beginning assembly language programmer.

ASSEMBLY LINES: THE BOOK - by Roger Wagner
A compilation of the first 18 issues of the Assembly Lines series. In addition, the text has been extensively edited and a unique encyclopedia-like appendix added. This appendix shows not only the basic details of each 6502 command, but also a brief discussion of its most common uses along with concise, illustrative listings.

HOW TO ENTER CALL - APPLE ASSEMBLY LANGUAGE LISTINGS
Call-APPLE, Volume IV, No.1, January 81.

MACHINE TOOLS
Call-APPLE in Depth, No. 1

BEGINNER´S GUIDE TO USING Merlin

By T. Petersen

Notes and demonstrations for the beginning Merlin programmer.

Introduction

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

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

A note of clarification:

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

In this section, "module" refers to a distinct computer program component of the Merlin system. There are four MODULES in the DOS 3.3 Merlin, five in the ProDOS Merlin:

1. The EXECUTIVE
2. The EDITOR
3. The ASSEMBLER
4. The LINKER
5. The COMMAND INTERPRETER (ProDOS version only)

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

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

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

EXECUTIVE CONTROL MODE
    Executive Module
    Command Interpreter (ProDOS version only)

EDITOR CONTROL MODE
    Editor Module
    Assembler Module
    Linking Loader

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

Input

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

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

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

Steps from the very beginning:

1. Boot the Merlin disk.

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

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

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

5. After return, the cursor once again drops down one line, a ´2´ appears and the cursor skips a space.

6. Now, hit the space bar once and type ´ORG´, space again, type ´$8000´, and hit RETURN.

The above step instruct the assembler to place the following program logically (with ORG) at $8000.

7. On line 3, do not space once after the line number. Type ´BELL´, space, ´EQU´, space, ´$FBDD´, RETURN.

This defines the label BELL to be equal to hex FBDD. This type (use) of a label is known as a constant. Wherever BELL appears in an expression, it will be replaced with $FBDD. Why don´t we just use ´$FBDD´? For one thing, ´BELL´ is easier to remember than ´$FBDD´ (making ´BELL´ in effect a mnemonic). Also, if the location of BELL were to change, all that needs changing is the ´EQU´ statement, and all the other ´$FBDD´s throughout the listing.

8. Line 4 - Type ´START´, space ´JSR´, space ´BELL´, space, ´;´ (semicolon), ´RING THE BELL´, RETURN. Semicolons are a convention often used within command lines to mark the start of comments.

9. Line 5 ´DONE´, space, ´RTS´, RETURN.

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

11. The screen should now appear like this:

```
1  *DEMO PROGRAM 1
2        ORG   $8000
3  BELL  EQU   $FBDD
4  START JSR   BELL         ;RING THE BELL
5  DONE  RTS
```

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

Field Number...

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

Two is reserved for opcodes, such as the Merlin pseudo-opcodes ORG and EQU, and the 6502 opcodes JSR and RTS.

Three is for operands, such as $8000, $FBDD and, in this case, BELL.

Four will contain comments (preceded by ";").

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

In summary, after the line numbers:

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

System Control and Text Entry Commands

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

The following paragraphs contain brief demonstrations for both system control and line editing.

All System and Entry commands are used in EDITOR Control Mode immediately after the ´:´ prompt.

CTRL-X, CTRL-C or a RETURN as the first character of a line exits the current [entry command] mode and returns the system to control mode when ADDing or INSERTing lines. CTRL-X or CTRL-C exits edit mode and returns the system to control mode after Editing lines.

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

Inserting and deleting lines in the source code are both simple operations. The following example will INSERT three new lines between the existing lines 4 and 5.

1. After the ´:´ prompt, type ´I´ for (INSERT), the number ´5´, and press RETURN. All inserted lines will precede (numerically) the line number specified in the command.

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

3. Repeat step 2.

4. Enter one space, type ´TYA´, and press RETURN.

   On the screen is the following:

```
:I5
   5 *
   6 *
   7     TYA
   8
```

5. Hit RETURN and the system reverts to CONTROL mode (´:´ prompt).

6. LIST the source code.

```
    :L
         1  *DEMO PROGRAM 1
         2  *
         3           ORG    $8000
         4  BELL     EQU    $FBDD
         5  *
         6  *
         7           TYA
         8  START    JSR    BELL         ;RING THE BELL
         9  END      RTS
```

The three new lines (5,6, and 7) have been inserted, and the subsequent original source lines (now lines 8 and 9) have been renumbered.

Using DELETE is equally easy.

1. In control mode, input ´D7´, and RETURN. Nothing new appears on the screen.

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

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

1. In control mode, input ´D5,6´ and RETURN.

2. LIST the source.

Lines 5 and 6 from the previous example, which contained the inserted asterisk comments, have been deleted, and the sub sequent lines renumbered. The listing appears the same as in the subsection on INPUT, Step 12.

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

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

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

   ```
   6 DONE   RTS
   ```

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

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

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

4. Repeat step 1 (above).

5. This time, type CTRL-I. Don´t move the cursor with the space bar or arrow keys. Type the word ´DONE´, and RETURN.

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

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

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

Assembly

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

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

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

Type N, and you will see:

```
ASSEMBLING
                    1    *DEMO PROGRAM 1
                    2
                    3              ORG  $8000
                    4    BELL      EQU  $FBDD
8000 20 DD FB       5    START     JSR  BELL      ;RING THE BELL
8003 60             6    DONE      RTS

--END ASSEMBLY, 4 BYTES, ERRORS: 0


SYMBOL TABLE - ALPHABETICAL ORDER

     BELL      =$FBDD          ?     DONE          =$8003
?    START     =$8000

SYMBOL TABLE - NUMERICAL ORDER

?    START     =$8000          ?   DONE            =$8003
     BELL      =$FBDD
```

If instead of completing the above listing, the system beeps and displays an error message, note the line number referenced in the message, and press RETURN until the "--END ASSEMBLY..." message appears. Then refer back to the subsection on INPUT and compare the listing with step 12. Look especially for elements in incorrect fields. Using the editing functions you´ve learned, change any lines in your listing which do not look like those in the listing in step 12 to what they should, then re-assemble.

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

To the left of the line numbers, beginning on line 5, is a series of numeric and alphabetic characters. This is the object code the opcodes and operands assembled to their machine language hexadecimal equivalents.

Left to right, the first group of characters is the routine´s starting address in memory (see the definition of ORG in the section entitled "Pseudo Opcodes Directives"). After the colon is the number ´20´. This is the one-byte hexadecimal code for the opcode JSR.

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

The next two bytes (each pair of hexadecimal digits is one byte) on line 5 bear a curious resemblance to the last group of characters on line 4; have a look. In line 4 of the source code we told the assembler that the label ´BELL´ EQUated with address $FBDD. In line 5, when the assembler encountered ´BELL´ as the operand, it substituted the specified address. The sequence of the high- and low-order bytes was reversed, turning $FBDD into DD FB, a 6502 microprocessor convention.

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

Saving and Running Programs

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

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

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

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

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

While writing either file to disk, MERLIN also displays the address parameter, and calculates and displays the length parameter. It´s a good practice to take note of these. Viewing the catalog will show that although the optional A$ and L$ parameters were displayed on the EXEC mode menu, they were not saved as part of the file names. If you´d prefer to have this information in the disk catalog, use the DOS RENAME command. Make sure no commas are included in the new file name.

Return to EDITOR mode (press ´E´). Next type ´GET $8000´. This command will tell Merlin to take the program you´ve just assembled and transfer it to the Apple´s main memory. Next, type ´MON´, RETURN and the monitor prompt ´*´ appears. Enter ´8000G´, RETURN. A beep is heard. The demonstration program was responsible for it. It works!

Now you can return to the EXEC by typing CTRL-Y and hitting RETURN.

Making Back-up Copies of MERLIN

The MERLIN diskette is unprotected and copies may be made using any copy utility. It is highly recommended that you use ONLY the BACK-UP copy of MERLIN in your daily work, and keep the original in a safe place. All files and also the side containing SOURCEROR.FP can be moved:

1) to any DOS 3.3 diskette using the FID utility program from Apple´s System Master Diskette;

2) to any ProDOS diskette using the FILER utility program from the ProDOS User´s Disk.

EXECUTIVE MODE

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

C:CATALOG (DOS 3.3)

When you press "C", the CATALOG of the current diskette will be shown. The word "COMMAND:" is then printed and MERLIN will let you enter a DOS command. This facility is provided primarily for locking and unlocking files. Unlike the LOAD SOURCE, SAVE SOURCE, and APPEND FILE commands, you must type the ".S" suffix when referencing a source file. Do not use it to load or save files. If you do not want to give a disk command, just hit RETURN. Use CTRL-X to cancel a partially typed command. If you type CTRL-C RETURN after "COMMAND:", you will be presented with the EXEC mode prompt "%". You can then issue any EXEC command such as "L" for LOAD SOURCE. This permits you to give an EXEC mode command while the catalog is still on the screen. In addition, if CTRL-C is typed at the "CATALOG pause" point, printing of the remainder of the catalog is aborted.

C:CATALOG (ProDOS)

When you press "C", you will be asked for the Pathname of the Directory you wish to catalog. Enter a pathname or press RETURN for the current directory and the CATALOG of the current directory will be shown. The EXEC mode prompt "%" is displayed after the catalog is shown. You can then issue any EXEC command such as "L" for LOAD SOURCE. This permits you to give an EXEC mode command while the catalog is still on the screen. In addition, if ANY key is typed during the CATALOG printing, the ProDOS catalog will pause until any other key is pressed.

If you enter a "1" as the first character of a pathname (or just 1<RETURN>) then the catalog will be sent to the printer in slot 1.

L:LOAD SOURCE

This is used to load a binary source file from disk. You will be prompted for the name of the file. You should not append ".S" since MERLIN does this automatically. If you have hit "L" by mistake, just hit RETURN twice and the command will be cancelled without affecting any file that may be in memory.

After a LOAD SOURCE (or APPEND SOURCE) command, you are automatically placed in the editor mode, just as if you had hit "E". The source will automatically be loaded to the correct address. Subsequent LOAD SOURCE or SAVE SOURCE com mands will display the last used filename, followed by a flashing "?". If you hit the "Y" key, the current file name will be used for the command. If you hit any other key (such as RETURN) the cursor will be placed on the first character of the filename, and you may type in the desired name. RETURN alone without typing a file name you will cancel the command.

S:SAVE SOURCE

Use this to save a binary source file to disk. As in the load command, you do not include the suffix ".S" and you can hit RETURN to cancel the command. NOTE: the address and length of the source file are shown on the MENU, and are for information only. You should not use these for saving; the assembler remembers them better than you can and sends them to DOS automatically. As in the LOAD SOURCE command above, the last loaded or saved filename will be displayed and you may type "Y" to save the same filename, or any key for a new file name.

A:APPEND FILE

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

D:DRIVE CHANGE (DOS 3.3)

When you hit "D", the drive used for saving and loading will change from one to two or two to one. The currently selected drive is shown on the menu. When MERLIN is first booted, the selected drive will be the one used by the boot. There is no command to specify slot number, but this can be accomplished by typing "C" for CATALOG which will display the current disks directory. Then give the disk command "CATALOG,Sn", where n is the slot number. This action will catalog the newly specified drive.

D:DISK COMMAND (ProDOS)

This allows yo to issue disk related commands. The following commands are available with the Merlin Interpreter:

```
PREFIX  pathname  (sets the prefix to pathname)
PFX     pathname  (shorthand for PREFIX)
BLOAD   pathname  [,A$....] (only hex addresses allowed)
BRUN    pathname  [,A$....] (only hex addresses allowed)
-       pathname  [,A$....] (only hex addresses allowed)
BSAVE   pathname,A$adrs,L$len
DELETE  pathname
LOCK    pathname
UNLOCK  pathname
RENAME  old_pathname,new_pathname
ONLINE        (shows the drives currently on line and
               their names)
```

A disk command returns to the disk command mode. You can then issue another disk command or just hit RETURN to go back to the menu.

When PREFIX, or PFX, is entered without a pathname, the PREFIX command sets the prefix to the "volume" part of the current prefix. For example, if the current prefix is /MERLIN/LIB and you type PFX <return> at the disk command prompt, the the prefix will revert to /MERLIN.

BLOAD, BRUN and "-" accept both BIN and SYS files. The difference between BRUN and "-" is in the state of the soft switches when control is passed to the program. BRUN leaves Merlin up; that is, auxiliary zero page and language card RAM are selected. The "-" command switches in the main zero page and the $D000-$FFFF roms. An RTS from such a program will return to Merlin. Most of the utility programs supplied with Melin (SOURCEROR, XREF, etc.) can be run by either method. You can use "-" (but NOT BRUN) to run programs such as the ProDOS FILER. However, such programs do not return to Merlin and the /RAM/ volume is left disconnected by this procedure.

E:ENTER ED/ASM

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

O:SAVE OBJECT CODE

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

NOTE: the object address shown is that of the program´s ORG (or $8000 by default) and not that of the actual current location of the assembled code (which is ordinarily $8000 in auxiliary memory). When using this command, you are asked for a name for the object file. Unlike the source file case, no suffix will be appended to this name.

Thus you can safely use the same name as that of the source file (without the ".S" of course). When this object code is saved to the disk its address will be the correct one, the one shown on the menu. When later you BLOAD or BRUN it, it will load at that address, which can be anything ($300,$8000, &c).

Q:QUIT (DOS 3.3)

This exits to BASIC. You may re-enter MERLIN by issuing the "ASSEM" command. This re-entry will be a warm start, which means it will not destroy the source file currently in memory. This exit can be used to give disk commands, test machine language programs, run BASIC programs, etc.

Q:QUIT (ProDOS)

This exits the Merlin Interpreter. You must specify the PREFIX for the next interpreter and then the pathname of the next interpreter, i.e. the one you are quitting to. In most cases this will be the /BASIC.SYSTEM interpreter.

R:READ TEXT FILE (DOS 3.3)

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

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

The READ TEXT FILE and WRITE TEXT FILE commands will include a "T." at the beginning of the filename you specify UNLESS you precede the filename with a space or any other character in the ASCII range of $20 to $40. This character will be ignored and not used by DOS in the actual filename.

The READ TEXT FILE and WRITE TEXT FILE commands are used to LOAD or CREATE "PUT" files, or to access files from other assemblers or text editors.

W:WRITE TEXT FILE (DOS 3.3)

This writes a MERLIN file into a text file instead of a binary file. The speed of the READ TEXT FILE and WRITE TEXT FILE commands is approximately that of a standard DOS BLOAD or BSAVE. The WRITE TEXT FILE routine does a VERIFY after the write.

@:SET DATE (ProDOS)

This allow you to set the current date for ProDOS. Note that this option does not set the date on a clock card. If you have a clock, the date stamping is automatic (provided you have a Thunderclock or have installed the requisite clock driver). The SET DATE provision is intended for people who do not have a clock. In that case, you may use this to set the currnet date and this date will then be used for date stamping. You may also just use this to check on the current date. Type RETURN alone to exit the SET DATE routine.

THE EDITOR

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

ABOUT THE EDITOR DOCUMENTATION

The editor documentation, as a whole, is broken into three major sections:

1) The Command Mode Commands
2) The Add/Insert Mode Commands
3) The Edit Mode Commands

For each of the commands, the documentation consists of three basic parts:

1) the name and syntax of the command,
2) examples of the use of each available syntax,
3) a description of the function of each command.

When the syntax for each command is given:

PARENTHESES () indicate a required value,
ANGLE BRACKETS <> indicate an optional value or character.
SQUARE BRACKETS [] are used to enclose comments about the commmand.

COMMAND MODE

For many of the COMMAND mode commands, only the first letter of the command is required, the rest being optional. This manual will show the required command characters in UPPER case and the optional ones in lower case.

Line Numbers in Command Mode

With some commands, you must specify a line number, a range of line numbers or a range list. A line number is just a number. A range is a pair of line numbers separated by a comma. A range list consists of several ranges separated by a slash ("/").

Line Number examples:

10	LINE #	[a single line number]
10,30	RANGE	[the range of lines 10 to 30]
10,30/50,60	RANGE LIST	[ranges 10 to 30 AND 50 to 60]

If a line number in a range exceeds the number of the last line in the source, the editor automatically adjusts the specified line to the last line number.

Delimited Strings (or d-strings)

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

```
Delimited string examples:
   ´this is a delimited string´
   "this is a delimited string"
   @this is another d-string@
```

Note that the slash "/" cannot be used as a delimiter since it is the character that delimits range lists in the editor.

Wild Card Characters in Delimited Strings

For all of the commands that use delimited strings (d-strings), the "^" character acts as a wild card character. Therefore, the d-string "Jon^s" is equivalent to the d-string "Jones" as well as "Jonas".

Upper and Lower Case Control

The Apple //e or Apple //c shift and caps lock keys work as you would expect in the control mode.

THE COMMAND MODE COMMANDS

Hex-Dec Conversion

```
        128 = $0080
        $80 = 128
```

If you type a decimal number (positive or negative) in the command mode, the hex equivalent is returned. If you type a hex number, prefixed by "$", the decimal equivalent is returned. All commands accept hex numbers.

NEW

```
        NEW                   [ only option for this cmd ]
```

Deletes the present source file in memory.

PR#

PR#(0-7)

```
        PR#1    [ can be used to send output to printer ]
        PR#3    [note: do not use for 80 col card ]
```

Same function as in BASIC. Mainly used for sending an editor or assembly listing to a printer. DO NOT use this to select an 80 - column card. NOTE: that PR# is automatically turned off after an ASM command, but not after a LIST or PRINT command.

Note that the PR# command can be used to send an assembly listing to the printer unformatted and without page breaks. If formatting and page breaks are desired use the PRTR command.

USER

```
     USER
     USER 1          [ example for use with XREF ]
     USER  "SOURCE"  [ example for use with PRINTFILER ]
```

This does a JSR $3F5. (That is the Applesoft ampersand vector location, which normally points to an RTS.) The designed purpose of this command is for the connection of the various utilities supplied with Merlin and for user defined printer drivers. (You must be careful that your printer driver does not use zero page addresses, except the I/O pointers and $60 -$6F, because this is likely to interfere with MERLIN´s heavy zero page usage). Several supplied utilities operate through the USER command.

TABS

```
TABS <number><, number><,...> <"tab character">
     TABS                [ clear all tabs ]
     TABS 10,20          [ set tabs to 10 & 20 ]
     TABS 10,20 " "      [ as above,space is tab char ]
```

This sets the tabs for the editor, and has no effect on the assembler listing. Up to nine tabs are possible. The default tab character is a space, but any may be specified. The assembler regards the space as the only acceptable tab character for the separation of labels, opcodes, and operands. If you don´t specify the tab character, then the last one used remains. Entering TABS and a RETURN will set all tabs to zero.

LENgth

```
     LEN                 [ only option for this cmd ]
```

This gives the length in bytes of the source file, and the number of bytes free.

Where

Where (line number)

```
     W 50                 [ where is line 50 in memory ]
     W 0                  [ where is end of source file ]
```

This prints in hex the location in memory of the start of the specified line. "Where 0" (or "W0") will give the location of the end of source.

MONitor

```
     MON                  [ only option with this cmd ]
```

This exits to the monitor. You may re-enter MERLIN at the executive level by either CTRL-C, CTRL-B or CTRL-Y. These re-establish the important zero page pointers from a save area inside MERLIN itself. Thus CTRL-Y will give a correct entry, even if you have messed up the zero page pointers while in the monitor. DOS is not connected when using this entry to the monitor.

You may also re-enter the editor directly with a 0G. This re-entry, unlike the others, will use the zero page pointers at $0A - $0F instead of the ones saved upon exit. Therefore, you must be sure that they have not been altered. For normal usage, however, one of the three CTRL's is to be used to re-enter MERLIN.

Note that when you exit to the monitor with this command, the RAM-based $D000-$FFFF memory is enabled, i.e. Merlin and it's symbol table (if any). If you want to examine the ROM memory that would ordinarily correspond to Applesoft and the F8 Monitor, you should quit Merlin using the normal QUIT command, and then enter the Monitor with the usual CALL-151 statement. Note: under ProDOS, this procedure will necessitate loading the BASIC.SYSTEM and Merlin Pro will be removed from memory.

TRuncON

```
     TRON                 [ only option for this cmd ]
```

When used as an immediate command, sets a flag which, during LIST or PRINT, will terminate printing of a line upon finding a space followed by a semicolon. It makes reading of source files easier on the Apple 40 column screen.

TRuncOFf

```
     TROF                    [ only option for this cmd ]
```

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

Quit

```
     Q                       [ only option for this cmd ]
```

Exits to EXEC mode.

ASM

```
        ASM                    [ only option for this cmd ]
```

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

NOTE: During the second pass of assembly, typing a CTRL-D will toggle the list flag, so that listing will either stop or resume. This will be defeated if a LST opcode occurs in the source, but another CTRL-D will override it. Assembly times are significantly faster with the listing turned off.

Delete

```
Delete (line number)
Delete (range)
Delete (range list)
        D 10                   [ deletes line number 10 ]
        D 10,32                [ deletes lines 10 through 32 ]
        D 20,30/10,12          [ deletes ranges of lines 10
                                 through 12 and 20 through 30 ]
```

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

Replace

```
Replace (line number)
Replace (range)
       R 30                    [ Delete 30 then goto Insert ]
       R 30,40                 [ Delete 30 to 40, then Insert ]
```

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

List

```
List
List (line number)
List (range)
List (range list)
       L                       [ list entire file ]
       L 20                    [ list line 20 only ]
       L 20,30                 [ list 20 through 30 ]
       L 20,30/40,42           [ list 20 through 30 and then
                                 list lines 40 through 42 ]
```

Lists the source file with line numbers. Control characters in source are shown in inverse, unless the listing is being sent to a printer or other nonstandard output device.

The listing can be aborted by CTRL-C or with "/" key. You may stop the listing by hitting the space bar and then advance a line at a time by hitting the space bar again. By holding down the space bar, the auto repeat feature of the Apple will result in a slow listing. Any other key will restart it. This space bar pause also works during assembly and the symbol table print out.

```
   [period]
                                [ only option for this cmd ]
```

Lists starting from the beginning of the last specified range. For example, if you type "L10,100",lines 10 to 100 will be listed. If you then use ".", listing will start again at 10 and continue until stopped (the end of the range is not remembered).

/

```
/ <line number>
        /                       [ start to list at last line
                                  listed ]
        /50                     [ start listing at line 50 ]
```

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

Print

```
Print
Print (line number)
Print (range)
Print (range list)
        P                       [ print entire file ]
        P50                     [ print line 50 only ]
        P50,100                 [ print lines 50 through 100 ]
        P1,10/20,30             [ print 1 through 10 and then
                                  print lines 20 through 30 ]
```

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

PRinTeR

```
PRinTeR (command)
        PRTR 1 ""               [ activate printer in slot 1 with
                                  no printer string ]
        PRTR 1 "<ctr>I80n"      [ as above, but add control I80N
                                  to initalize the printer ]
        PRTR 1 "" Page Title [ printer in slot 1, no control
                                   string, "Page Title" is the
                                   page header ]
        PRTR 3                  [ send formatted listing to 80
                                  column screen ]
        PRTR 8                  [ send output through the vector
                                  at $3F5 ]
```

This command is for sending a listing to a printer with page headers and provision for page boundary skips. (See the section on configuration for details on setting up default parameters.) The entire syntax of this command is:

```
PRTR slot#  "(string)" <page header>
```

If the slot number used is more than seven, a JSR $3F5 (ampersand vector) is done and it is expected that the routine there will connect a printer driver by putting its address in locations $36-$37.

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

THE INITIALIZATION STRING MAY NOT BE OMITTED IF A PAGE HEADER IS TO BE USED. If no special string is required by the printer, use a null string (in which case a carriage return will be used). Examples of initialization strings are CTRL-Q for IDS printers, CTRL-I80N for most Apple printer cards or ESC 1 to place an Okidata printer in correspondence mode. Note that you must use CTRL-O prior to typing ESC so that you don´t go into escape mode.

PRTR 0 (no strings required here) will allow you to see where the page breaks occur. If the 80 column card is in use in slot 3, then use PRTR 3 for this.

No output is sent to the printer until a LIST, PRINT, or ASM command is issued.

Find

```
Find (d-string)
Find (line number) <d-string>
Find (range) <d-string>
Find (range list) <d-string>
     F "A String"           [ finds lines with "A String" ]
     F 10 "STRING"          [ finds "STRING" if in line 10]
     F 10,20 "HI"           [ finds lines in range of 10
                              through 20 that contain "HI"]
     F 10,20/50,99 "HI"     [ finds lines that contain "HI"
                              in range of 10 through 20 and
                              50 through 99 ]
```

This lists those lines containing the specified string. It may be aborted with CTRL-C or "/" key.

Change

```
Change (d-string d-string)
Change (line number) <d-string d-string>
Change (range) <d-string d-string>
Change (range list) <d-string d-string>
     C "hello"goodbye    [ finds "hello" and if told to do
                           so will change it to "goodbye"]
     C 50 "hello"bye     [ changes in line 50 only ]
     C 50,100 "Hello"BYE [ changes lines 50 through 100 ]
     C 50,60/65,66 "AND"OR [changes in lines 50 through 60
                            and lines 65 and 66]
```

This changes occurrences of the first d-string to the second d-string. The d-strings must have the same delimiters. For example, to change occurrences of "speling" to "spelling" throughout the range 20,100, you would type C20,100 "speling"spelling. If no range is specified the entire source file is used.

Before the change operation begins, you are asked whether you want to change "all" or "some". If you select "some" by hitting the "S" key, the editor stops whenever the first string is found and displays the line as it would appear with the change.

(Change continued)

If you then hit the "Y" key the change will be made. If you press the "RETURN" key the change will not be made. In reality, typing any control character such as ESCAPE, RETURN or any others will result in the change not being made. Any other key, such as "Y" (or even "N") will accept the change. CTRL-C or "/" key will Abort the change process.

COPY

```
COPY (line number) TO (line number)
COPY (range) TO (line number)
        COPY 10 TO 20          [ copies line 10 to just before
                                 line 20 ]
        COPY 10,20 TO 30       [ copies lines 10 through 20 to
                                 just before line 30 ]
```

This copies the line number or range to just ´above´ the specified number. It does not delete anything.

MOVE

```
MOVE (line number) TO (line number)
MOVE (range) TO (line number)
        MOVE 10 TO 20          [ Move line 10 to just before 20]
        MOVE 10,20 TO 30       [ Move lines 10 through 20 to
                                 just before line 30 ]
```

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

FW (Find Word)

```
FW (d-string)
FW (line number) <d-string>
FW (range) <d-string>
FW (range list) <d-string>
        FW "LABEL"          [ find all lines with "LABEL" ]
        FW20 "LABEL"        [ try to find "LABEL" in 20 ]
        FW20,30 "PTR"       [ find all lines between 20 and
                              30 that contain "PTR" ]
        FW20,30/50,99 "PTR" [ find all lines between 20 and
                              30 and between 50 and 99 that
                              contain the word "PTR" ]
```

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

Therefore, FW"CAT" will find:

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

but will not find CATALOG or SCAT.

CW (Change word)

```
Change (d-string d-string)
Change (line numbers) <d-string d-string>
Change (range) <d-string d-string>
Change (range list) <d-string d-string>
        CW "PTR"PRT         [ change all "PTR"s to "PRT"s ]
        CW 20 "PTR"PRT      [ as above but only in line 20 ]
        CW 20,30 "PTR"PRT   [ do the same as the above but
                              for lines 20 through 30 ]
        CW 1,9/20,30 "PTR"PRT
                            [ same as above but include lines
                              1 through 9 in the range ]
```

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

EW (Edit word)

```
EW (d-string)
EW (line number) <d-string>
EW (range) <d-string>
EW (range list) <d-string>
        EW "PTR              [ edit lines with "PTR" ]
        EW 10 "PTR           [ edit 10 if "PTR" is there]
        EW 10,20 "PTR        [ as above, but 10 through 20 ]
        EW 1,5/10,20 "PTR    [ as above, but include 1 to 5 ]
```

This is to EDIT as FW is to FIND.

Edit

```
Edit
Edit (line number)
Edit (range)
Edit (range list)
Edit (d-string)
Edit (line number) <d-string>
Edit (range) <d-string>
Edit (range list) <d-string>
        Edit                 [ edit ALL lines ]
        Edit 10              [ edit line 10 ]
        Edit 10,20           [ edit lines 10 through 20 ]
        Edit 1,5/9,20        [ edit lines 1 through 5 and
                               lines 9 through 20 ]
        Edit "START"         [ edit all lines that contain the
                               d-string "START" ]
        Edit 10 "START"      [ edit line 10 IF "START" is
                               is in the line ]
        Edit 10,20 "END"     [ edit all lines in range of 10
                               through 20 that contain "END" ]
        Edit 10,20/50,100 "LABEL"
                             [ edit all lines in range of 10
                               through 20 and 50 through 100
                               that contain the d-string
                               "LABEL" ]
```

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

TEXT

```
    TEXT                [ only option for this cmd ]
```

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

FIX

```
    FIX                 [ only option for this cmd ]
```

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

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

ProDOS NOTE: Fix MUST be use and the ´fixed´ file resaved, for text files transported from DOS 3.3 to ProDOS using CONVERT.

VIDeo

VIDeo (slot)

```
    VID 3               [ select video in slot 3 ]
```

This command is designed to select or deselect an 80 column board. The default condition can be selected using the configure program included on the Merlin diskette. This is similar to the use of PR# in BASIC. DO NOT use PR# to select an 80 column board! PR# is designed for selection of a printer ONLY. An 80 column board in slot 3 for example, should be selected by typing, from the editor: VIDEO 3.

It is deselected by ESC CTRL-Q for the Apple //e 80 column board or by ESC 0 for the Videx Ultra Term.

VAL

```
VAL "expression"
        VAL "PTR                [ return value of label "PTR" ]
        VAL "LABEL"             [ Gives the address (or value) of
                                  LABEL for the last assembly
                                  done or "unknown label" if not
                                  found.  ]
        VAL "$1000/2"           [ returns $0800 ]
        VAL "%1000"             [ returns $0008 ]
```

This will return the value of the expression as the assembler would compute it. All forms of label and literal expressions valid for the assembler are valid for this command. Note that labels while have the value given them in the most recent assembly.

GET

```
GET ( obj adrs )
        GET                     [ put object in main memory at
                                  the address specified in the
                                  sources ORG ]
        GET $4000               [ put object at location $4000
                                  in main memory ]
```

This moves the object code from its location in auxiliary memory to main memory at the specified address. The address must be above the existing source file, if any, and it will not be allowed to clobber DOS. You can do a NEW if you want to load it lower in memory than allowed, but remember to save your source first. You cannot use this to put the object code at memory locations lower than $901 but you can go to the monitor afterwords and use it to move it to any desired location. Any such move using the monitor may, however, destroy you source or other data valuable to Merlins operation. Caution should be used!

The GET command does not check if a valid object code has been assembled.

SWAP

SWAP

```
        SWAP                    [ only option for this cmd ]
```

This swaps the source file in main memory with one in auxiliary memory. It can be used before GET to hide the source while testing a program. SWAP is very useful with the DOS 3.3 version of Merlin, since you can work on your assembly language program with Merlin in the aux bank of memory. When you have a sucessful assembly, you can use SWAP to hide your source in the aux bank, Quit Merlin to return to Applesoft and then run or test your assembly program. You end up with two complete computers, a "Merlin computer" and an "Applesoft computer."

NOTE: The GET and SWAP commands could be dangerous and may result in loss of source. If you use this and GET to test a program from the monitor you MUST either protect ALL zero page locations, or return to the EXEC mode by using by ^Y, ^C or ^B. Returning to the editor first by using 0G will NOT protect zero page locations.

Protecting zero page locations is best done by saving and restoring any such locations used. Particularly important are the pointers at locations $0A through $0F. Another method would be to switch zero pages and switch back before returning to Merlin. Merlin uses the AUX zero page. Disaster may result if zero pages are switched and fail to be switched back.

The SWAP command will overwrite the object code if the source extends to $8000 or beyond. Type "WO" to check this before using SWAP. SWAP will also overwrite the XREF program, or other USER programs. Conversely, loading some USER progams or other utilities may destroy the swapped source file or the main one.

Issuing the ASM command automatically deletes any file SWAPped into auxiliary memory.

ADD/INSERT MODES

Add

```
     A                        [ only option for this cmd ]
```

The Add command places you in the ADD mode, and acts much like entering additional BASIC lines with auto line numbering. To exit from ADD mode, hit RETURN as the FIRST character of a line. You may also exit the ADD mode by CTRL-X or CTRL-C which also cancels the current line.

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

Insert

```
Insert (line number)
     I 20                     [ inserts lines "above" line 20 ]
```

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

Add/Insert Mode Editing Commands

All of the commands described in the "Edit Mode Commands" section of this manual will work in the ADD mode as well as the INSERT mode. The only exception is CTRL-R, which during a "real" edit restores the line being edited to its original condition. Since any line being Added or Inserted did not previously exist it cannot be restored. Hence, CTRL-R does nothing.

EDIT MODE

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

The EDIT commands and functions are very similar, but not identical to those in Neil Konzen′s GPLE and RWPI′s A.C.E. All commands except CTRL-R are available in ADD and INSERT modes.

Edit Mode Commands

Control-I (insert)

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

Note that the keyboard′s TAB key issues a control-I and therefore will issue an Insert command to Merlin.

Control-D (delete)

Deletes the character under the cursor.

Delete Key

This is a backwards delete. It deletes the character preceding the cursor.

Control-F (find)

Finds the next occurrence of the character typed after the CTRL-F. To move the cursor to the next occurrence on the line, press the character key again.

Control-O (insert special)

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

Control-P (do ***´s)

If entered as the first character of a line gives 32 *´s. If entered as any other character of the line, gives 30 spaces bordered by *´s. Note that these asterisks replace any characters on the line you are editing when you press CTRL-P.

Control-C or Control-X (cancel)

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

Control-B (go to line begin)

Places the cursor at the beginning of the line.

Control-N (go to line end)

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

Control-R (restore line)

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

Control-Q (accept line to cursor position)

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

Return (RETURN key)

> Accepts the complete line as it appears on the screen and fetches the next line to be edited, or goes to the command mode.

The Editor´s Handling of Strings and Comments with Spaces

When entering strings or comments in the Add/Insert or Edit modes, you will sometimes find the editor inserting additional spaces. The editor will, however, remove the added spaces when the line is terminated with a RETURN.

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

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

In a line such as LDA #´ ´, you can prevent the extra tabbing by entering the line with a space before the first quote (LDA # ´ ´), then use the cursor control keys to move back and delete the extra space.

THE ASSEMBLER

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

ABOUT THE ASSEMBLER DOCUMENTATION

The assembler documentation is broken into three main sections:

1) Preliminary Definitions,
2) Assembler Syntax Conventions,
3) Assembler Pseudo Opcode Descriptions.

The last two sections are each further broken down into the following:

Asssembler Syntax Conventions:

1) Number Format
2) Source Code Format
3) Expressions Allowed by the Assembler
4) Immediate Data Syntax
5) 6502 and 65C02 Addressing Modes
6) Sweet 16 Opcodes

Assembler Pseudo Opcode Descriptions:

1) Assembler Directives
2) Formatting Pseudo Ops
3) String Data Pseudo Ops
4) Data and Storage Allocation Pseudo Ops
5) Miscelaneous Pseudo Ops
6) Conditional Pseudo Ops
7) Pseudo Ops for Macros
8) Variables

The Assembler Syntax Conventions illustrate the syntax of a line of assembly code, the proper method to specify numbers and data, how to construct assembler expressions and the proper syntax to use to specify the different addressing modes allowed by the 6502 or 65C02 microprocessors. This section should be understood prior to using the assembler, otherwise it is will be difficult to determine the acceptable methods to, for instance, construct a proper expression.

The Assembler Pseudo Opcode Descriptions illustrate the functions of the many Merlin Pseudo Ops, the correct syntax to use and examples of each Pseudo Ops use.

Preliminary Definitions

The type of operand for almost all of Merlin´s pseudo ops and the 6502 and 65C02 microprocessors can be grouped into one of four categories:

1) Expressions
2) Delimited Strings (d-strings)
3) Data
4) Filenames or Pathnames

EXPRESSIONS

Expressions are defined in the Assembler Syntax Conventions section of the manual.

DELIMITED STRINGS

Delimited Strings are defined in the EDITOR section of the manual, but that definition is repeated here for continuity.

> Several of the Pseudo Opcodes, and some of the 6502 and 65C02 opcodes allow their operand to be a string. Any such string must be delimited by a non-numeric character other that the slash (/) or comma (,). Such a string is called a "d-string". The usual delimiter is a single or double quote mark (" or ´).

(Delimited Strings continued)

Examples:

```
"this is a d-string"
´this is another d-string"
@another one@
Zthis is one delimited by an upper case zZ
"A"
´A´
```

Note that delimited strings used as the object of ANY 6502 or 65C02 opcode MUST be enclosed in single or double quotes. If not, the assembler will interpret the d-string to be a label, expression or data instead.

Take special note that some of the pseudo ops as well as the 6502 and 65C02 opcodes use the delimiter to determine the hi-bit condition of the resultant string. In such cases the delimiter should be restricted to the single or double quote.

DATA

Data is defined as raw hexadecimal data composed of the digits 0..9 and the letters A..F.

FILENAMES (DOS 3.3 only)

Filenames are defined as the name of a DOS 3.3 file without any delimiters, e.g. no quotes surrounding the name. Source file names are suffixed with ".S". Text files, USES files and PUT files are prefixed with "T.". The applicable suffix or prefix should not be used as part of the filename.

PATHNAMES (ProDOS only)

Pathnames are defined as ProDOS pathnames and as such are restricted to the definition of pathnames as described in the ProDOS USER´S MANUAL. Pathnames as used by Merlin do not have delimiters, e.g. no quotes surrounding the pathname. Source, USES, and PUT pathnames are suffixed with ".S". This suffix should not be used as part of the pathname.

ASSEMBLER SYNTAX CONVENTIONS

Source Code Format

Syntax of a Source Code Line

A line of source code typically looks like:

```
     LABEL    OPCODE  OPERAND       ;COMMENT
```

and a few real examples:

```
  1  START       LDA  #50           ;THIS IS A COMMENT
  2  *    THIS IS A COMMENT ONLY LINE
  3                                 ;TABBED BY EDITOR
```

A line containing only a comment can begin with "*" as in line 2 above. Comment lines starting with ";", however, are accepted and tabbed to the comment field as in 3 above. The assembler will accept an empty line in the source code and will treat it just as a SKP 1 instruction (see the section on pseudo opcodes), except that the line number will be printed.

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

Source Code Label Conventions

The maximum allowable LABEL length is 13 characters, but more than 8 will produce messy assembly listings. A label must begin with a character at least as large, in ASCII value, as the colon, and may not contain any characters less, in ASCII value, than the number zero. Note that periods (.) are not allowed in labels since the period is used to specify the logical OR in expressions.

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

Source Opcode and Pseudo Opcode Conventions

The assembler examines only the first 3 characters of the OPCODE (with certain exceptions such as macro calls and the Sweet 16 POPD). For example, you can use PAGE instead of PAG (because of the exception, the fourth letter should not be a D, however). The assembler listing will not look well with an opcode longer than five characters unless there is no operand.

Operand and Comment Length Conventions

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

NUMBER FORMAT

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

```
      100         $64          %1100100          %01100100
```

As indicated by the last binary number, leading zeros are ignored.

Immediate Data vs. Addresses

In order to instruct the assembler to interpret a number as immediate data as opposed to an address, the number should be prefixed with a "#". The "#" here stands for "number" or "data". For example:

```
LDA #100        LDA #$64      LDA #%1100100
```

These three instructions will all LOAD the Accumulator with the number 100, decimal.

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

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

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

Use of Decimal, Hexadecimal or Binary Numbers

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

```
     DA    $1
     DA    $A
     DA    $64
     DA    $3E8
     DA    $2710
```

is a good deal more mysterious than its decimal equivalent:

```
     DA    1
     DA    10
     DA    100
     DA    1000
     DA    10000
```

Similarly,

```
     ORA #$80
```

is less informative than

```
     ORA #%10000000
```

which sets the hi-bit of the number in the accumulator.

EXPRESSIONS ALLOWED BY THE ASSEMBLER

To make the syntax accepted and/or required by the assembler clear, we must define what is meant by an "expression".

Primitive Expressions

Expressions are built up from "primitive expressions" by use of arithmetic and logical operations. The primitive expressions are:

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

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

Arithmetic and Logical Operations in Expressions

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

Building Expressions

Expressions are built using the primitive expressions defined above, either with or without arithmetic and/or logical operations. This means that expressions can take the form of primitives or primitives operated on by other primitives using the arithmetic and logical operators.

Some examples of legal expressions are:

#01	(primitive expression = 1)
#$20	(primitive expression = 32 dec)
LABEL	(primitive consisting of a label)
#"A"	(primitive consisting of letter "A")
*	(primitive = current value of PC)

The following are examples of more complex expressions

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

Parentheses and Precedence in Expressions

Parentheses are not normally allowed in expressions. They are not used to modify the precedence of expression evaluation. All arithmetic and logical operations are evaluated left to right (2+3*5 would assemble as 25 and not 17).

Parentheses are used to retrieve a value from the memory location specified by the value of the expression within the parentheses, much like indirect addressing. This use is restricted to certain pseudo ops, however. For example:

```
    DO  ($300)
```

will instruct the assembler to generate code if the value of memory location $300, at the time of assembly, is non-zero.

Example of Use of Assembler Expressions

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

```
COMPARE    LDX    #EODATA-DATA-1
LOOP       CMP    DATA,X
           BEQ    FOUND      ;found
           DEX
           BPL    LOOP
           JMP    REJECT     ;not found
DATA       HEX    CACFC5D9
EODATA     EQU    *
```

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

IMMEDIATE DATA SYNTAX

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

#<expression produces the low byte of the expression

#>expression produces the high byte of the expression

#expression also gives the low byte (the 6502 does not accept 2-byte DATA)

#/expression is optional syntax for the high byte of the expression

6502 ADDRESSING MODES

The assembler accepts all the 6502 and 65C02 opcodes with standard mnemonics. It also accepts BLT (Branch if Less Than) and BGE (Branch if Greater or Equal) as pseudonyms for BCC and BCS, respectively.

There are 12 addressing modes available. The appropriate MERLIN syntax for these are:

	Syntax	Example
Implied	OPCODE	CLC
Accumulator	OPCODE	ROR
Immediate (data)	OPCODE #expr	ADC #$F8
		CMP #"M"
		LDX #>LABEL1-LABEL2-1
Zero page (address)	OPCODE expr	ROL 6
Indexed X	OPCODE expr,X	LDA $E0,X
Indexed Y	OPCODE expr,Y	STX LAB,Y
Absolute (address)	OPCODE expr	BIT $300
Indexed X	OPCODE expr,X	STA $4000,X
Indexed Y	OPCODE expr,y	SBC LABEL-1,Y
Indirect	JMP (expr)	JMP ($3F2)
Preindexed X	OPCODE (expr,X)	LDA (6,X)
Postindexed Y	OPCODE (expr),Y	STA ($FE),Y

Special Forced Non-Zero Page Addressing

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

```
LDA  $10   assembles as zero page (2 bytes) while,
LDA: $10   assembles as non-zero page (3 bytes).
```

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

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

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

Sweet 16 Opcodes

The assembler accepts all Sweet 16 opcodes with the standard mnemonics. The usual Sweet 16 registers R0 to R15 do not have to be "equated" and the "R" is optional. For the SET opcode, either a space or a comma may be used between the register and the data part of the operands; that is, SET R3,LABEL is equivalent to SET R3LABEL. It should be noted that the NUL opcode is assembled as a one-byte opcode (the same as HEX 0D) and not a two byte skip as this would be interpreted by ROM Sweet 16. This is intentional, and is done for internal reasons.

Note: The Sweet 16 opcodes will not be recognized by the assembler unless the SW pseudo opcode has been previously assembled. This pseudo op will enable assembly of Sweet 16.

65C02 and 65802 Opcodes

The assembler will assemble 65C02 source code as well as 65802 source code. The XC pseudo opcode activates these features. This opcode is discussed in the following section on Pseudo ops.

ASSEMBLER PSEUDO OPCODE DESCRIPTIONS

Directives

EQU (=)   (EQUate)

```
Label  EQU  expression
Label  =    expression   (alternate syntax)
        START EQU $1000       [ equate START to $1000]
        CHAR  EQU "A"         [ equate CHAR to ascii val of A]
        PTR    =   *          [ PTR equals present PC]
```

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

Note that Labels are CASE SENSITIVE. Therefore, the assembler will consider the following labels as different labels:

```
                START     [ upper case label]
                Start     [ mixed case label]
                start     [ lower case label]
```

EXT   (EXTernal label)

```
 label  EXT                 [ label is external labels name]
         PRINT  EXT         [ define PRINT as external ]
```

This pseudo op defines a label as an external label for use by the Linker. The value of the label, at assembly time, is set to $8000, but the final value is resolved by the linker. The symbol table will list the label as having the value of $8000 plus its external reference number (0-$FE). See the LINKER section of the manual for more information on this opcode.

ENT (ENTry label)

```
 label  ENT
        PRINT  ENT          [ define PRINT as entry label ]
```

This pseudo op will define the label in the label column as an ENTRY label. An entry label is a label that may be refered to as an EXTernal label by another REL code module. The true address of an entry label will be resolved by the LINKER.

The REL code module being written, or assembled, may refer to the ENT label just as if it were an ordinary label. It can be EQU´d, jumped to, branched to, etc.

The symbol table listing will print the relative address of the label and will flag it as an "E".

See the LINKER section of the manual for more information on this opcode.

ORG (set ORiGin)

```
 ORG expression
 ORG
        ORG  $1000          [ start code at $1000 ]
        ORG  START+END      [ start at value of expression]
        ORG                 [ re-ORG ]
```

Establishes the address at which the program is designed to run. It defaults to $8000. Ordinarily there will be only one ORG and it will be at the start of the program. If more than one ORG is used, the first one establishes the BLOAD address, while the second actually establishes the origin. This can be used to create an object file that would load to one address though it may be designed to run at another address.

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

(ORG continued)

ORG without an operand is accepted and is treated as a "REORG" type command. It is intended to be used to reestablish the correct address pointer after a segment of code which as a different ORG. (When used in a REL file, all labels in a section between an "ORG address" and an "ORG noaddress" are regarded as absolute addresses. This is meant ONLY to be used in a section to be moved to an explicit address.)

Example of ORG without an operand:

```
                1            ORG  $1000
1000: A0 00     2            LDY  #0
1002: 20 21 10  3            JSR  MOVE       ;"MOVE" IS
1005: 4C 12 10  4            JMP  CONTINUE   ;NOT LISTED.
                5            ORG  $300       ;ROUTINE TO
0300: 8D 08 C0  6  PAGE3     STA  MAINZP     ;BE MOVED
0303: 20 ED FD  7            JSR  COUT
0306: 8D 09 C0  8            STA  AUXZP
0309: 60        9            RTS
                10           ORG             ;REORG
1012: A9 C1     11 CONTINUE  LDA  #"A"
1014: 20 00 03  12           JSR  PAGE3
```

REL (RELocatable code module)

REL

REL [only option for this opcode]

This opcode instructs the assembler to generate code files compatable with the relocating linker. This opcode must occur prior to the use or definition of any labels. See the LINKER section of this manual for more information on this opcode.

OBJ (set OBJect)

```
 OBJ expression
        OBJ  $4000          [ use of hex address ]
        OBJ  START          [ use with a label ]
```

The OBJ opcode is accepted only prior to the start of the code and it only sets the division line between the symbol table and object code areas in memory (which defaults to $8000). The OBJ address is accepted only if it lies between $4000 and $BFE0. Most people should never have to use this opcode. If the REL opcode is used then OBJ is disregarded. If DSK is used then you can, but should not have to, set OBJ to $BFE0 to maximize the space for the symbol table.

PUT (PUT a text file in assembly)

```
 PUT filename
     DOS 3.3 EXAMPLES
        PUT  SOURCEFILE     [PUT´s file T.SOURCEFILE]
        PUT  !SOURCE        [PUT´s file SOURCE]
        PUT  !SOURCE,D2     [PUT´s file SOURCE from drive 2]
     ProDOS EXAMPLES
        PUT  SOURCEFILE     [PUT´s file SOURCEFILE]
        PUT  /PRE/SOURCE    [PUT´s file SOURCE from DIR PRE]
```

"PUT filename" reads the named file and "inserts" it at the location of the opcode.

DOS 3.3 NOTE: Drive and slot parameters are accepted in the standard DOS syntax. The "filename" specified must be a text file with the "T." prefix. If it doesn´t have the "T." prefix in the disk catalog, the "filename" specified must start with a character less than "@". This tells MERLIN to look for a file without the "T." prefix. The "!" character can be used for this purpose. For example:

```
Disk file name =  T.SOURCE CODE     [name in catalog]
PUT file name  =  SOURCE CODE       [name in PUT opcode]

Disk file name = SOURCE CODE        [name in catalog]
PUT file name  = !SOURCE CODE       [name in PUT opcode]
```

(PUT continued)

ProDOS NOTE: Drive and slot parameters are not accepted, pathnames must be used. Note that the above name conventions do not apply to ProDOS, since all source files under ProDOS are text files.

NOTE: "Insert" refers to the effect on assembly and not to the location of the source. The file itself is actually placed just following the main source. These files are in memory only one at a time, so a very large program can be assembled using the PUT facility.

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

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

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

USE (USE a text file as a macro library)

```
 USE filename
         USE  T.MACRO LIBRARY       [DOS 3.3 example]
         USE  !MACROS               [DOS 3.3, no "T." prefix]
         USE  T.MACROS,S5,D1        [DOS 3.3 with slot/drive]
         USE  /LIB/MACROS           [ProDOS pathname]
```

This works as does a PUT but the file is kept in memory. It is intended for loading a macro library that is USEd by the source file.

VAR (setup VARiables)

```
 VAR expr;expr;expr...
         VAR  1;$3;LABEL        [ set up VAR's 1,2 and 3 ]
```

This is just a convenient way to equate the variables]1 -]8. "VAR 3;$42;LABEL" will set]1 = 3,]2 = $42, and]3 = LABEL. This is designed for use just prior to a PUT. If a PUT file uses]1 -]8, except in PMC (or >>>) lines for calling macros, there MUST be a previous declaration of these.

SAV (SAVe object code)

```
 SAV filename
         SAV  FILE              [ ProDOS or DOS 3.3 syntax ]
         SAV  /OBJ/PROG         [ ProDOS pathname syntax ]
```

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

This pseudo-opcode provides a means of saving portions of a program having more than one ORG. It also enables the assembly of extremely large files. After a save, the object address is reset to the last specification of OBJ or to $8000 by default.

Files saved with the SAVe command will be saved to BLOAD to the correct address.

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

TYP (set ProDOS file type for DSK and SAV) (ProDOS only)

```
 TYP expression
         TYP  $00               [ no file type ]
         TYP  $06               [ binary file type ]
```

This sets the file type to be used by the DSK or SAV opcodes. The default is the BIN type. Valid file types are 0,6,$F0-$F7, and $FF (no type, BIN, CMD, user defined, SYS).

DSK (assemble directly to DiSK)

```
 DSK filename (or pathname for ProDOS)
         DSK PROG               [ DOS 3.3 or ProDOS ]
         DSK /OBJ/PROG          [ ProDOS pathname example ]
```

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

NOTE: Files intended for use with the linking loader MUST be saved with the DSK pseudo op, see the REL opcode.

END (END of source file)

```
 END
         END                    [ only option for this opcode ]
```

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

DUM (DUMmy section)

```
 DUM expression
         DUM $1000              [ start DUMmmy code at $1000 ]
         DUM LABEL              [ start code at value of LABEL]
         DUM END-START          [ start at val of END-START ]
```

This starts a section of code that will be examined for value of labels but will produce no object code. The expression must give the desired ORG of this section. It is possible to re-ORG such a section using another DUMMY opcode or using ORG. Note that although no object code is produced from a dummy section, the text output of the assembler will appear as if code is being produced.

DEND (Dummy END)

```
 DEND
         DEND                  [ only option for this opcode ]
```

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

Sample usage of DUM and DEND:

```
         ORG  $1000

IOBADRS  =    $B7EB

         DUM  IOBADRS
IOBTYPE  DFB  1
IOBSLOT  DFB  $60
IOBDRV   DFB  1
IOBVOL   DFB  0
IOBTRCK  DFB  0
IOBSECT  DFB  0
         DS   2          ;pointer to DCT
IOBBUF   DA   0
         DA   0
IOBCMD   DFB  1
IOBERR   DFB  0
ACTVOL   DFB  0
PREVSL   DFB  0
PREVDR   DFB  0
         DEND

START    LDA  #SLOT
         STA  IOBSLOT
*   And so on
```

FORMATTING PSEUDO OPS

LST ON/OFF (LiSTing control)

```
 LST ON or OFF
        LST ON                 [ turn listing on ]
        LST OFF                [ turn listing off ]
        LST                    [ turn listing on, optional ]
```

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

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

NOTE: CONTROL-D from the keyboard toggles this flag during the second pass.

EXP ON/OFF/ONLY (macro EXPand control)

```
 EXP ON or OFF or ONLY
        EXP ON                 [ macro exapand on ]
        EXP OFF                [ print only macro call ]
        EXP ONLY               [ print only generated code ]
```

EXP ON will print an entire macro during the assembly. The OFF condition will print only the PMC pseudo-op. EXP defaults to ON. This has no effect on the object coded generated. EXP ONLY will cause expansion of the macro to the listing omitting the call line and end of macro line. (if the macro call line is labeled, however, it is printed.) This mode will print out just as if the macro lines were written out in the source.

LSTDO or LSTDO OFF (LiST DO OFF areas of code)

```
 LSTDO
 LSTDO  OFF
        LSTDO                  [ list the DO OFF areas ]
        LSTDO  OFF             [ don't list DO OFF areas ]
```

This opcode causes the listing of DO OFF areas of code to be printed in listings or not to be printed.

PAU (PAUse)

```
 PAU
        PAU                    [ only option for this opcode ]
```

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

PAG (new PAGe)

```
 PAG
        PAG                    [ only option for this opcode ]
```

This sends a formfeed ($8C) to the printer. It has no effect on the screen listing even when using an 80-column card.

AST (send a line of ASTerisks)

```
 AST expression
        AST 30                 [ send 30 asterisks to listing ]
        AST NUM                [ send NUM asterisks ]
```

This sends a number of asterisks (*) to the listing equal to the value of the operand. The number format is the usual one, so that AST10 will send (decimal) 10 asterisks, for example. The number is treated modulo 256 with 0 being 256 asterisks!

SKP (SKiP lines)

```
 SKP expression
        SKP  5                 [ skip 5 lines in listing ]
        SKP  LINES             [ skip "LINES" lines in listing]
```

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

TR ON/OFF (TRuncate control)

```
TR ON or OFF
        TR  ON                 [ limit object code printing ]
        TR  OFF                [ don't limit object code print]
```

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

DAT (DATe stamp assembly listing) (ProDOS only)

```
 DAT
        DAT                    [ only option for this opcode ]
```

This prints the current date and time on the second pass of the assembler. Available only in ProDOS Merlin.

CYC (calculate and print CYCle times for code)

```
 CYC
 CYC OFF
 CYC AVE

        CYC                    [ print opcode cycles & total ]
        CYC OFF                [ stop cycle time printing ]
        CYC AVE                [ print cycles & average ]
```

This opcode will cause a program cycle count to be printed during assembly. A second CYC opcode will cause the accumulated total to go to zero. CYC OFF causes it to stop printing cycles. CYC AVE will average in the cycles that are underterminable due to branches, indexed and indirect addressing.

(CYC continued)

The cycle times will be printed (or displayed) to the right of the comment field and will appear similar to any one of the following:

```
5 ,0326       or       5' ,0326      or      5'',0326
```

The first number displayed (the 5 in the example above) is the cycle count for the current instruction. The second number displayed is the accumulated total of cycles in decimal.

A single quote after the cycle count indicates a possible added cycle, depending on certain conditions the assembler cannot forsee. If this appears on a branch instruction then it indicates that one cycle should be added if the branch occurs. For non-branch instructions, the single quote indicates that one cycle should be added if a page boundary is crossed.

A double quote after the cycle count indicates that the assembler has determined that a branch would be taken and that the branch would cross a page boundary. In this case the extra cycle is displayed and added to the total.

The CYC opcode will also work for the extra 65C02 opcodes in Merlin. It will not work for the additional 65C02 opcodes present in the Rockwell 65C02 (i.e. RMB#, SMB#, BBR# and BSS#). These opcodes are not supported by Merlin, except when USEing the ROCKWELL macro library. All of these unsupported opcodes are 5-cycle instructions with the usual possible one or two extra cycles for the branch instructions BBS and BBR.

The CYC opcode will also work for the 65802 opcodes, but it will NOT add the extra cycles required when M=0 or when X=0.

STRING DATA PSEUDO OPS

General notes on String Data and String Delimiters

Different delimiters have different effects. Any delimiter less than (in ASCII value) the single quote (´) will produce a string with the high-bits on, otherwise the high-bits will be off. For example, the delimiters !"#$%& will produce a string in "negative" ASCII, and the delimiters ´()+? will produce one in "positive" ASCII. Usually the quote (") and single quote (´) are the delimiters of choice, but other delimiters provide the means of inserting a string containing the quote or single quote as part of the string. Example delimiter effects:

```
"HELLO"                 [ negative ascii, hi bit set ]
!HELLO!                 [ negative ascii, hi bit set ]
#HELLO#                 [ negative ascii, hi bit set ]
&HELLO&                 [ negative ascii, hi bit set ]
!ENTER "HELLO"!         [ string with embedded quotes ]
´HELLO´                 [ positive ascii, hi bit clr ]
(HELLO(                 [ positive ascii, hi bit clr ]
´ENTER "HELLO"´         [ string with embedded quotes ]
```

All of the opcodes in this section, except REV, also accept hex data after the string. Any of the following syntaxes are acceptable:

```
ASC "string"878D00
FLS "string",878D00
DCI "string",87,8D,00
STR "STRING",878D00
INV "string",878D00
```

ASC (define ASCii text)

```
ASC d-string
     ASC  "STRING"        [ negative ascii string ]
     ASC  ´STRING´        [ positive ascii string ]
     ASC  "Bye,Bye",8D    [ negative with added hex bytes]
```

This puts a delimited ASCII string into the object code. The only restriction on the delimiter is that it does not occur in the string itself.

DCI (Dextral Character Inverted)

```
 DCI d-string
        DCI   "STRING"          [ neg ascii, except for the "G" ]
        DCI   ´STRING´          [ pos ascii, except for the "G" ]
        DCI   ´Hello´,878D      [ pos with two added hex bytes ]
```

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

INV (define INVerse text)

```
 INV d-string
        INV   "STOP!"           [ neg ascii, inverse on printing]
        INV   ´END´,878D        [ positive, added bytes ]
```

This puts a delimited string in memory in inverse format.

FLS (define FLaShing text)

```
 FLS d-string
        FLS   "The End"         [ neg ascii, flash on printing]
        FLS   ´The End´,8D00 [ pos,flash with added bytes ]
```

This puts a delimited string in memory in flashing format.

REV (REVerse)

```
 REV d-string
        REV   "Insert"          [ neg ascii, reversed in mem ]
        REV   ´Insert"          [ same as above but positive ]
```

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

REV "DISK VOLUME"

gives EMULOV KSID (delimiter choice as in ASC). HEX data may NOT be added after the string terminator.

STR (define a STRing with a leading length byte)

```
 STR d-string
        STR  "/PATH/"        [ pos ascii, (ProDOS pathname?)]
        STR  "HI"            [ result= 02 C8 C9 ]
        STR  'HI',8D         [ result= 02 48 49 8D ]
```

This puts a delimited string into memory with a leading length byte. Otherwise it works the same as the ASC opcode. Note that following HEX bytes, if any, are NOT counted in the length. This facility is mainly intended for use with ProDOS which uses this type of data extensively.

DATA AND STORAGE ALLOCATION PSEUDO OPS

DA or DW (Define Address or Define Word)

```
 DA expression  or DW expression
        DA $FDF0              [ results: F0 FD in mem ]
        DA 10,$300            [ results: 0A 00 00 03 ]
        DW LAB1,LAB2          [ example of use with labels ]
```

This stores the two-byte value of the operand, usually an address, in the object code, low-byte first.

These two pseudo ops also accept multiple data separated by commas (such as DA 1,10,100).

DDB (Define Double-Byte)

```
 DDB expression
    DDB  $FDED+1              [ results: FD EE in memory ]
    DDB  10,$300              [ results: 00 0A 03 00 ]
```

As above with DA, but places high-byte first. DDB also accepts multiple data (such as DDB 1,10,100).

DFB or DB (DeFine Byte or Define Byte)

```
 DFB expression  or DB expression
        DFB  10               [ results: 0A in memory ]
        DFB  $10              [ results: 10 in memory ]
        DB   >$FDED+2         [ results: FD in memory ]
        DB   LAB              [ example of use with label ]
```

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

DFB continued

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

For example:

```
DFB $34,100,LAB1-LAB2,%1011,>LAB1-LAB2
```

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

```
34 64 DE 0B 09
```

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

HEX (define HEX data)

```
 HEX hex-data
      HEX  0102030F         [ results: 01 02 03 0F in mem ]
      HEX  FD,ED,C0         [ results: FD ED C0 in memory ]
```

This is an alternative to DFB which allows convenient insertion of hex data. Unlike all other cases, the "$" is not required or accepted here. The operand should consist of hex numbers having two hex digits (for example, use 0F, not F). They may be separated by commas or may be adjacent. An error message will be generated if the operand contains an odd number of digits or ends in a comma, or as in all cases, contains more than 64 characters.

DS (Define Storage)

```
 DS expression
 DS expressioul, expression2
 DS \
 DS \,expression2
        DS  10                  [ zero out 10 bytes of mem ]
        DS  10,$80              [ put $80 in 10 bytes of mem ]
        DS  \                   [ zero mem to next memory page ]
        DS  \,$80               [ put $80 in mem to next page ]
```

This reserves space for string storage data. It zeros out this space if the expression is positive. DS 10, for example, will set aside 10 bytes for storage.

Because DS adjusts the object code pointer, an instruction like DS-1 can be used to back up the object and address pointers one byte.

The first alternate form of DS, with two expressions, will fill expressionl bytes with the value of (the low byte of) expression2, provided expression2 is positive. If expression2 is missing 0 is used for the fill.

The second alternate form, "DS \", will fill memory (with 0's) until the next memory page. The "DS \,expression2" form does the same but fills using the low byte of expression2.

Notes for REL files and the Linker

The "\" options are intended for use mainly with REL files and work slightly differently with these files. Any "DS \" opcode occurring in a REL file will cause the linker to load the next file at the first available page boundary, and to fill with 0's or the indicated byte. Note that, for REL files, the location of this code has NO EFFECT on its action. To avoid confusion, you should put this code at the end of a file.

MISCELANEOUS PSEUDO OPS

KBD (define label from KeyBoarD)

```
label KBD
label KBD d-string
         OUTPUT  KBD             [ get value of OUTPUT from kbd ]
         OUTPUT  KBD "send to printer"
                                 [ promt with the d-string for
                                   the value of OUTPUT ]
```

This allows a label to be equated from the keyboard during assembly. Any expression may be input, including expressions referencing previously defined labels, however a BAD INPUT error will occur if the input cannot be evaluated.

The optional delimited string will be printed on the screen instead of the standard "Give value for LABEL:" message. A colon is appended to the string.

LUP

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

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

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

which will assemble as:

```
ASL
ASL
ASL
ASL
```

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

(LUP continued)

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

```
]A      =    0
        LUP $FF
]A      =    ]A+1
        DFB ]A
        --^
```

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

The maximum LUP value is $8000 and the LUP opcode will simply be ignored if you try to use more than this.

NOTE: the above use of incrementing variables in order to build a table WILL NOT work if used within a macro. Program structures such as this must be included as part of the main program source.

CHK (place CHecKsum in object code)

```
 CHK
      CHK                    [ only option for this opcode ]
```

This places a checksum byte into object code at the location of the CHK opcode. This is usually placed at the end of the program and can be used by your program at runtime to verify the existence of an accurate image of the program in memory.

ERR (force ERRor)

```
 ERR expression
 ERR \expression
      ERR  $80-($300)        [ error if $80 not in $300 ]
      ERR  *-1/$4100         [ error if PC > $4100 ]
      ERR  \$5000            [ error if REL code address
                               exceeds $5000 ]
```

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

(ERR continued)

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

```
ERR  *-1/$9600
```

NOTE: The above example would only alert you that the program is too long, and will not prevent writing above $9600 during assembly, but there can be no harm in this, since the assembler will cease generating object code in such an instance. The error occurs only on the second pass of the assembly and does not abort the assembly.

Another available syntax is: ERR ($300)-$4C

which will produce an error on the first pass and abort assembly if location $300 does not contain the value $4C.

Notes for REL Files and the ERR Pseudo Op

The "ERR \expression" syntax gives an error on the second pass if the address pointer reaches expression or beyond. This is equivalent to "ERR *-1/expr", but it when used with REL files, it instructs the linker to check that the last byte of the current module does not extend to expression or beyond (expression must be absolute). If the linker finds that the current module DOES extend beyond expression, linking will abort with a message "Constraint error:" followed by the value of expression in the ERR opcode. You can see how this works by trying to link the PI file to an address over $81C. Note that the position of this opcode in a REL file has no bearing on its action, so that it is best to put it at the end.

SW (SWeet 16 opcodes)

SW

```
SW                    [ only option for this opcode ]
```

This enables Sweet 16 opcodes. If SW (similarly for XC) is not selected then those opcode names can be used for macros. Thus, if you are not using Sweet 16, you can use macros named ADD, SUB, etc.

XC (eXtended 65C02 and 65802 opCodes)

```
 XC
       XC                    [ enable the 65C02 option ]
       XC (twice in a row)   [ enable the 65802 option ]
```

This enables the extra 65C02 opcodes. If used twice, the 65802 codes can also be assembled. Note that some of the "long" 65802 addressing codes are not enabled since they do nothing useful on the 65802.

Note that the XC pseudo op will not enable the extended BIT opcodes used on the Rockwell 65C02 chip. There is, however, a macro library file included on the Merlin disk that can be USEd to implement these additional codes.

MX (long status Mode of 65802)

```
 MX  expression
       MX  %00               [ M & X 16 bit modes are on ]
       MX  %10               [ M mode on, X mode off ]
       MX  %01               [ X mode on, M mode off ]
       MX  3                 [ M & X 16 bit modes are off ]
```

This pseudo op is used to inform Merlin of the intended status of the "long" status of the 65802 processor. It functions only when the assembler is in the 65802 mode, i.e. when two consectutive XC opcodes have been given. The assembler cannot determine if the processor is in 16 bit memory mode (M status bit=0) or 16 bit index register mode (X status bit=0). The purpose of the MX opcode is to inform the assembler of the current status of these bits.

Three of the above examples use binary expressions as the operand of the MX opcode. Note that any valid expression may be used as long as it is within the range of 0-3.

This opcode MUST be used when using Merlin's 65802 capabilities to inform the assembler of the proper mode to use in order to insure proper assembly of immediate mode commands (such as LDA #expression, etc.).

USR (USeR definable op-code)

USR optional expressions

```
        USR expression    [ examples depend on definition]
```

This is a user definable pseudo-opcode. It does a JSR $B6DA. This location will contain an RTS after a boot, a BRUN MERLIN or BRUN BOOT ASM. To set up your routine you should BRUN it from the EXEC command after CATALOG. This should just set up a JMP at $B6DA to the main routine and then RTS.

The following flags and entry points may be used by your routine:

```
USRADS    = $B6DA  ;must have a JMP to your routine
PUTBYTE   = $E5F6  ;see below
EVAL      = $E5F9  ;see below
PASSNUM   = $2     ;contains assembly pass number
ERRCNT    = $1D    ;error count
VALUE     = $55    ;value returned by EVAL
OPNDLEN   = $BB    ;contains combined length of
                   ;operand and comment
NOTFOUND  = $FD    ;see discussion of EVAL
WORKSP    = $280   ;contains the operand and
                   ;comment in positive ASCII
```

Your routine will be called by the USR opcode with A=0, Y=0 and carry set. To direct the assembler to put a byte in the object code, you should JSR PUTBYTE with the byte in A.

(USR continued)

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

If your routine needs to evaluate the operand, or part of it, you can do this by a JSR EVAL. The X register must point to the first character of the portion of the operand you wish to evaluate (set X=0 to evaluate the expression at the start of the operand). On return from EVAL, X will point to the character following the evaluated expression. The Y register will be 0, 1, or 2 depending on whether this character is a right parenthesis, a space, or a comma or end of operand.

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

Appropriate locations for your routine are $300-$3CF and $8A0-$8FF. You must not write to $900.

You may use zero page locations $60-$6F, but should not alter other locations. Also, you must not change any thing from $226 to $27F, or anything from $2C4 to $2FF. Upon return from your routine (RTS), the USR line will be printed (on the second pass).

(USR continued)

When you use the USR opcode in a source file, it is wise to include some sort of check (in source) that the required routine is in memory. If, for example, your routine contains an RTS at location $310 then:

```
    ERR ($310)-$60
```

will test that byte and abort assembly if the RTS is not there. Similarly, if you know that the required routine should assemble exactly two bytes of data, then you can (roughly) check for it with the following code:

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

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

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

CONDITIONAL PSEUDO OPS

DO (DO if true)

```
 DO expression
        DO  0               [ turn assembly off ]
        DO  1               [ turn it on ]
        DO  LABEL           [ if LABEL<>0 then on ]
        DO  LAB1/LAB2       [ if LAB1<LAB2 then off ]
        DO  LAB1-LAB2       [ if LAB1=LAB2 then off ]
```

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

It is also useful for sources designed to generate slightly different code for different situations. For example, if you are designing a program to go on a ROM chip, you would want one version for the ROM and another with small differences as a RAM version for debugging purposes. Conditionals can be used to create these different object codes without requiring two sources.

Similarly, in a program with text, you may wish to have one version for Apples with lower case adapters and one for those without. By using conditional assembly, modification of such programs becomes much simpler, since you do not have to make the modification in two separate versions of the source code.

Every DO should be terminated somewhere later by a FIN and each FIN should be preceded by a DO. An ELSE should occur only inside such a DO/FIN structure. DO/FIN structures may be nested up to eight deep (possibly with some ELSE's between). If the DO condition is off (value 0), then assembly will not resume until its corresponding FIN is encountered, or an ELSE at this level occurs. Nested DO/FIN structures are valuable for putting conditionals in MACROS.

ELSE (ELSE do this)

```
ELSE
       ELSE                   [ only option for this opcode ]
```

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

IF (IF so then do)

```
IF char,]var    (IF char is the first character of ]var)
       IF (,]1                [ if first char of ]1 is "("
                                then assemble following code]
       IF ",]TEMP             [ if first char is ", assem ]
       IF "=]1                [ alternate use with "=" ]
```

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

```
IF "=]1
```

could be used to test if the first character of the variable]1 is a double quote (") or not, perhaps needed in a macro which could be given either an ASCII or a hex parameter.

FIN (FINish conditional)

```
FIN
       FIN                    [ only option for this opcode ]
```

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

EXAMPLE OF THE USE OF CONDITIONAL ASSEMBLY:

```
* Macro "MOV", moves data from ]1 to ]2
     MOV       MAC
               LDA  ]1
               STA  ]2
               <<<

* Macro "MOVD", moves data from ]1 to ]2 with many available
* syntaxes
     MOVD      MAC
               MOV  ]1;]2
               IF   (,]1           ;Syntax MOVD (ADR1),Y;????
               INY
               IF   (,]2           ;  MOVD (ADR1),Y;(ADR2),Y
               MOV  ]1;]2
               ELSE                ;  MOVD (ADR1),Y;ADR2
               MOV  ]1;]2+1
               FIN
               ELSE
               IF   (,]2           ;Syntax MOVD ????;(ADR2),Y
               INY
               IF   #,]1           ;  MOVD #ADR1;(ADR2),Y
               MOV  ]1/$100;]
               ELSE                ;  MOVD ADR1;(ADR2),Y
               MOV  ]1+1;]2
               FIN
               ELSE                ;Syntax MOVD ????;ADR2
               IF   #,]1           ;  MOVD #ADR1;ADR2
               MOV  ]1/$100;]2+1
               ELSE                ;  MOVD ADR1;ADR2
               MOV  ]1+1;]2+1
               FIN                 ;MUST close ALL
               FIN                 ;conditionals, Count DOs
               FIN                 ;& IFs, deduct FINs.  Must
               <<<                 ;yield zero at end.

     * Call syntaxes supported by MOVD:

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

MACRO PSEUDO OPS

MAC (begin MACro definition)

Label MAC

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

EOM (<<<)

```
EOM
<<<   (alternate syntax)
```

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

PMC (>>>) (macro-name)

```
PMC macro-name
>>> macro-name    (alternate syntax)
macro-name        (alternate syntax 2)
```

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

VARIABLES

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

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

It is possible to use variables for backwards branching, using the same label at numerous places in the source. This simplifies label naming for large programs and uses much less space than the equivalent once-used labels. For example:

```
 1         LDY #0
 2 ]JLOOP  LDA TABLE,Y
 3         BEQ NOGOOD
 4         JSR DOIT
 5         INY
 6         BNE ]JLOOP      ;BRANCH TO LINE 2
 7 NOGOOD  LDX #-1
 8 ]JLOOP  INX
 9         STA DATA,X
10         LDA TBL2,X
11         BNE ]JLOOP      ;BRANCH TO LINE 8
```

LOCAL LABELS

A local label is any label beginning with a colon. A local label is "attached" to the last global label and can be referred to by any line from that global label to the next global label. You can then use the same local label in other segments governed by other global labels. You can choose to use a meaningless type of local label such as :1, :2, etc., or you can use meaningful names such as :LOOP, :EXIT, and so on.

Example of local labels:

```
1    START  LDY #0
2           LDX #0
3    :LOOP  LDA (JUNK),Y        ;:loop is local to start
4           STA (JUNKDEST),Y
5           INY
6           CPY #100
7           BNE :LOOP           ;branch back to :LOOP in 3
8    LOOP2  LDY #0
9    :LOOP  LDA (STUFF),Y       ;:loop is now local to loop2
10          STA (STUFFDEST),Y
11          INY
12          CPY #100
13          BNE :LOOP           ;branch back to :LOOP in 9
14          RTS
```

Some restrictions on use of local labels:

Local labels cannot be used inside macros. You cannot label a MAC, ENT or EXT with a local label and you cannot EQUate a local label. The first label in a program cannot be a local label.

Local Labels, Global Labels and Variables

There are three distinct types of labels used by the assembler. Each of these are identified and treated differently by Merlin.

```
Global Labels : labels not starting with "]" or ":"
Local labels  : labels beginning with ":"
Variables     : labels beginning with "]"
```

(Local Labels, Global Labels and Variables continued)

Note that local labels do not save space in the symbol table, while variables do. Local labels CAN be used for forward and backward branching, while variables cannot. Good programming practice dictates the use of local labels as branch points, variables for passing data.

MACROS

Why Macros?

Macros represent a shorthand method of programming that allows multiple lines of code to be generated from a single statement, or Macro call. They can be used as a simple means to eliminate repetative entry of frequently used program segments, or they can be used to generate complex portions of code that the programmer may not even understand!

Examples of the first type are presented throughout this manual and in the T.MACRO LIBRARY file (/LIB/MACROS.S on the ProDOS disk). Examples of the second, more complex type, can be found in the T.FP MACROS (/LIB/FPMACROS.S on the ProDOS disk) and in the T.RWTS MACROS library found on the DOS 3.3 disk.

Macros can also be used to simulate unimplemented opcodes (available on the 6502) or to simulate the Rockwell 65C02 extended bit related opcodes, as in the T.ROCKWELL MACROS file (/LIB/ROCKWELL.S on the ProDOS disk.)

Macros literally allow you to write your own language and then turn that language into machine code with just a few lines of source code. Some people even take great pride in how many bytes of source code they can generate with a single Macro call!

How Does a Macro Work?

A macro is simply a user named sequence of assembly language statements, with general purpose operands. You define the macro in a general way, and when you use it, via a macro call, you "fill in the blanks" left when you defined it. Here's a short example:

```
MAC    SWAP     ;define a macro named SWAP
       LDA ]1   ;load accum with variable ]1, first blank
       STA ]2   ;store accum in location ]2, second blank
       <<<      ;this signals the end of the macro
```

In this example the "blanks" refered to previously are the variables]1, and]2. When you call the SWAP macro you provide a parameter list that "fills in" variables]1 and]2. What actually happens is the assembler substitutes the parameters you provide at assembly time for the variables. The order of substitution is determined by the parameter's place in the parameter list and the location of the corresponding variable in the macro definition. Here's how SWAP would be called and then filled in:

```
        SWAP  $00;$01
         |     |   |________{$01 takes place of ]2, 2nd parm}
         |     |
         |     |___________ {$00 takes place of ]1, 1st parm}
         |
         |_________________ { macro being called }
```

then, the macro will be "expanded" into assembly code,

```
        SWAP  $00;$01
        LDA   $00           {$00 in place of ]1}
        STA   $01           {$01 in place of ]2}
```

It is very important to realize that ANYTHING used in the parameter list will be substituted for the variables. For example:

```
        SWAP  #"A";DATA
```

would result in the following:

```
        SWAP  #"A";DATA
        LDA   #"A"
        STA   DATA
```

You can get even fancier if you like:

```
        SWAP  #"A";(STRING),Y
        LDA   #"A"
        STA   (STRING),Y
```

As illustrated, the substitution of the user supplied parameters for the variables is quite literal. It is quite possible to get into trouble this way also, but Merlin will inform you, via an error message, if you get too carried away. One common problem encountered is forgetting the difference between immediate mode NUMBERS and ADDRESSES. The following two macro calls will do quite different things:

```
        SWAP 10;20
        SWAP #10;#20
```

The first stores the contents of memory location 10 (decimal) into memory location 20 (decimal). The second macro call will attempt to store the NUMBER 10 (decimal) in the NUMBER 20! What has happened here is that an illegal addressing mode was attempted. The second macro call would be expanded into something like this (if it were possible):

```
        SWAP   #10;#20          ;call the SWAP macro
        LDA    #10              ;nothing wrong here
        STA    #20              ;woops! can´t do this!
*** BAD ADDRESS MODE ***        ;Merlin will let you know!
```

In order to use the macros provided with Merlin, or to write your own, study the macro in question and try to visualize how the required parameters would be substituted. With a little time and effort you´ll be using them like a PRO (pun intended).

Defining a Macro

A macro definition begins with the line:

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

with Name in the label field. Its definition is terminated by the pseudo-op EOM or <<<. The label you use as Name cannot be referenced by anything other than a valid Macro call: NAME, PMC NAME or >>> NAME.

Forward reference to a macro definition is not possible, and would result in a NOT MACRO error message. That is, the macro must be defined before it is called by NAME, PMC or >>>.

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

Labels inside macros are updated each time the macro NAME, PMC or >>> NAME is encountered.

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

Nested Macros

Macros may be nested to a depth of 15.

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

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

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

```
LDA LOC
STA DEST
```

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

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

A more common form of nesting is illustrated by these two macro definitions:

```
CH   EQU $24
POKE MAC
     LDA #]2
     STA ]1
     <<<
HTAB MAC
     >>> POKE.CH;]1
     <<<
```

The HTAB macro could then be used like this:

```
     HTAB 20            ;htab to column 20 decimal
```

and would generate the following code:

```
     LDA  #20           ;]2 in POKE macro
     STA  CH            ;]1 in POKE macro, 1st parm
                        ; in HTAB macro
```

MACRO names may also be put in the opcode column, without using the PMC or >>>, with the following restriction: The macro name cannot be the same as any regular opcode or pseudo opcode, such as LDA, STA, ORG, EXP, etc. Also, it cannot begin with the letters DEND or POPD.

Note that the PMC or >>> syntax is not subject to this restriction.

Special Variables

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

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

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

```
      MACRO DEFINITION           RESULTANT CODE EXAMPLE

TEMP      EQU       $10          SWAP.$6;$7;TEMP  ;macro call
          MAC
          LDA       ]1              LDA  $06
          STA       ]3              STA  TEMP
          LDA       ]2              LDA  $07
          STA       ]1              STA  $06
          LDA       ]3              LDA  TEMP
          STA       ]2              STA  $07
          <<<

          >>>       SWAP.$6;$7;TEMP
          >>>       SWAP.$1000;$6;TEMP
```

This program segment swaps the contents of location $6 with that of $7, using TEMP as a scratch depository, then swaps the contents of $6 with that of $1000.

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

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

The assembler will accept some other characters in place of the period (as per examples) or space between the macro name and the expressions in a PMC statement. You may use any of these characters:

```
. / , - (
```

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

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

MACRO DEFINITION

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

     >>>  MUV.(PNTR),Y;DEST
     >>>  MUV.#3;FLAG,X
```

with the resultant code from the above two Macro calls being:

```
>>>  MUV.(PNTR),Y;DEST    ;macro call
LDA  (PNTR),Y             ;substitute first parm
STA  DEST                 ;substitute second parm
```

and,

```
>>> MUV.#3;FLAG,X        ;macro call
LDA  #3                  ;substitute first parm
STA  FLAG,X              ;substitute second parm
```

It will also accept:

```
MACRO DEFINITION              RESULTANT CODE EXAMPLE

PRINT  MAC                    PRINT."Example"
       JSR  SENDMSG           JSR  SENDMSG
       ASC  ]1                ASC  "Example"
       BRK                    BRK
       <<<
```

Some additional examples of the PRINT macro call:

```
>>>  PRINT.!"quote"!
>>>  PRINT.´This is an example´
>>>  PRINT."So´s this, understand?"
```

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

Macro Libraries and the USES Pseudo Op

There are a number of macro libraries on the Merlin disk. These libraries are examples of how one could set up a library of often used macros. The requirements for a file to be considered a macro library are:

1) Only Macro definitions and label definitions exist in the file,
2) The file is a text file,
3) If it is a DOS 3.3 library, the file name must be prefixed with "T.",
4) The file must be accessable at assembly time (it must be on an available disk drive or "online").

The macro libraries included with Merlin include:

DOS 3.3	ProDOS	Macro Libary functions
T.FPMACROS	FPMACROS.S	- Allow easy access to Applesoft floating point math routines
T.MACROS	MACROS.S	- Often used macros for general use
T.ROCKWELL	ROCKWELL.S	- Implements extended bit related opcodes on the Rockwell 65C02
T.SENDMSG	SENDMSG.S	- A macro that allows easy printing from machine language
T.RWTS	<none>	- Allow easy access to DOS 3.3´s RWTS disk routines

Any of these macro libraries may be included in an assembly by simply including a USES pseudo op with the appropriate library name. There is no limit to the number of libraries that may be in memory at any one time, except for available memory space. See the documentation on the USES pseudo op for a discussion on its use in a program.

THE LINKER

Why a Linker?

The linking facilities built into Merlin offer a number of advantages over assemblers without this capability:

1) Extremely large programs may be assembled in one operation, over 41000 bytes long,

2) Large programs may be assembled much more quickly with a corresponding decrease in development time,

3) Libraries of subroutines (for disk access, graphics, screen/modem/printer drivers, etc.) may be developed and linked to any Merlin program,

4) Programs may be quickly re-assembled to run at any address,

With a linker you can write portions of code that perform specific tasks, say a general disk I/O handler, and perform whatever testing and debugging is required. When the code is correct, it is assembled as a REL file and placed on a disk. Whenever you need to write a program that uses disk I/O you won´t have to re-write or re-assemble the disk I/O portion of your new program. Just link your general disk I/O handler to your new program and away you go. This technique can be used for a variety of often used subroutines.

Wouldn´t a PUT file or Macro USES library serve the same purpose? A PUT file comes the closest tọ duplicating the utility of REL files and the linker, but there are a few rather large drawbacks for certain programs. First, using a PUT file to add a general purpose subroutine would result in much slower assembly. Second, any label definitions contained in the PUT file would be global within the entire program. With a REL file only labels defined as ENTry in the REL file (and EXTernal in the current file) would be shared by both programs. There is no chance for duplicate label errors when using the linker. Consider the following simple example:

An REL file has been assembled that drives a plotter. There are six entry points into the driver: PENUP, PENDOWN, NORTH, SOUTH, EAST, WEST. To further illustrate the value of a linker, assume the driver was written by a friend who has moved 2000 miles from you. Your job is to write a simple program to draw a box. The code would look something like this:

```
            REL                         ;RELOCATABLE CODE
PENUP       EXT                         ;EXTERNAL LABEL
PENDOWN     EXT                         ;ANOTHER ONE
NORTH       EXT
SOUTH       EXT
EAST        EXT
WEST        EXT

BOX         LDY  #00                    ;INITIALIZE Y
            JSR  PENDOWN                ;GET READY TO DRAW
:LOOP       JSR  NORTH                  ;MOVE UP
            INY                         ;INC COUNTER
            CPY  #100                   ;100 MOVES YET?
            BNE  :LOOP                  ; NOTICE LOCAL LABEL
            LDY  #00                    ;INIT Y AGAIN
:LOOP2      JSR  EAST                   ;NOW MOVE TO RIGHT
            INY
            CPY  #100
            BNE  :LOOP2                 ;FINISH MOVING RIGHT
* YOU GET THE IDEA, DO SOUTH, THEN WEST, AND DONE!
```

This simple sample program illustrates some of the power of RELocatable, linked files. Your program doesn't have to concern itself with conflicts between its' and the REL files labels, you don't concern yourself with the location of the EXTernal labels, your program listing is only 30 to 40 lines and it is capable of drawing a box on a plotter!

Some common examples of REL files that may not be readily apparent are found in Apple Pascal. The Turtlegraphics Unit, the Applestuff Unit, and with Apple Fortran the Run-Time libary are all examples of REL files.

Let´s look at another example that illustrates points 1 and 2 above. This time you are writing a data base program. You have broken the program down into 6 modules, all of which are REL files:

1) User interface
2) ISAM file system
3) Sort subsystem
4) Search subsystem
5) Report generator
6) Memory management subsystem

You would first design and write the User interface for your program. This would then be assembled and stored as a REL file. Next, the ISAM file system is written and de-bugged. You would then link the two modules together to see how they worked together. Next, you would complete the Sort, the Search, and all the rest. In fact, by using REL files, and documenting the ENTry points and their conditions, six different people could be working simultaneously on the same project and need no more from one another than the ENT labels!

To illustrate point 2, assume that the six modules are all coded as PUT files and that the resulting program was 40k bytes long (that´s 160 disk sectors or 80 disk blocks). The time it would take to assemble and cross reference such a large program would be measured in hours or days. Changing one byte in the source code would require a complete re-assembly and a quite a wait! By assembling each section independently as REL files and then linking them, the one byte change would require assembly of only one module in the 40k program. In short, with REL files and a linker, changes to large programs can be made quickly and efficiently, greatly speeding the program development process.

About the Linker Documentation

There are three pseudo opcodes that deal directly with relocatable modules and the linking process. These are:

REL - Informs the assembler to generate relocatable files
EXT - Defines a label as external to the current file
ENT - Defines a label in the current file as accessable to other REL files.

There are two other pseudo opcodes that behave differently when used in a REL file, relative to a normal file. These are:

DS - Define Storage opcode,
ERR- Force an ERRor opcode.

Each of these five pseudo opcodes will be defined or redefined in this section as they pertain to REL files. Also, an Editor command unique to REL files will also be defined: LINK.

In order to use the Linker, the files to be linked must be specifed. The linker uses a file containing the names of the files to be linked for this purpose. The format of this "linker name file" differs from DOS 3.3 and ProDOS. These differences will be illustrated here.

The Linker documentation will make no additional attempts to educate the user as to when (or when not) to use REL files.

Pseudo Opcodes for Use with Relocatable Code Files

REL (generate a RELocatable code file)

REL [only options for this opcode]

This opcode instructs the assembler to generate a relocatable code file for subsequent use with the relocating linker.

This MUST occur prior to definition of any labels. You will get a BAD "REL" error if not. REL files are incompatible with the SAV pseudo op and with the EXEC mode's object code save command. To get an object file to the disk you MUST use the DSK opcode for direct assembly to disk.

There are additional illegal opcodes and procedures that are normal with standard files.

An ORG at the start of the code is not allowed.

Multiplication, division or logical operations can be appled to absolute expressions but not relatives one.

Examples of absolute expressions are:

- An EQUate to an explicit address,
- The difference between two relative labels,
- Labels defined in DUMMY code sections.

Examples of relative expressions that are not allowed are:

- Ordinary labels,
- Expressions that utilize the PC, like: LABEL=*.

The starting address of an REL file, supplied by the assembler, is $8000. Note that this address is a fictional address, since it will later be changed by the linker. It is for this reason that no ORG opcode is allowed.

There are some restrictions involving use of EXTernal labels in operand expressions. No operand can contain more than one external. For operands of the following form:

```
#>expression    or    >expression
```

where the expression contains an external, the value of the expression must be within 7 bytes of the external labels´ value. For example:

```
LDA #>EXTERNAL+8          [ illegal expression ]
DFB >EXTERNAL-1           [ legal expression ]
```

Object files generated with the REL opcode are given the file type LNK under ProDOS. This is the type that will show if the disk is cataloged by Merlin. This type is file type $F8.

EXT (define a label EXTernal to the current REL module)

```
 label  EXT
        PRINT  EXT              [ define label PRINT as EXT ]
```

This defines the label in the label column as an external label. Any external label must be defined as an ENTry label in its own REL module, otherwise it will not be reconciled by the linker (the label would not have been found in any of the other linked modules). The EXTernal and ENTry label concepts are what allows REL modules to communicate and use each other as subroutines, etc.

The value of the label is set to $8000 and will be resolved by the linker. In the symbol table listing, the value of an external will be $8000 plus the external reference number ($0-$FE) and the symbol will be flagged with an "X".

ENT (define a label as an ENTry label in a REL code module)

```
 label  ENT
        PRINT  ENT              [ define label PRINT as ENTry ]
```

This defines the label in the label column as an ENTry label. This means that the label can be referred to as an external label. This facility allows other REL modules to use the label as if it were part of the current REL module. If a label is meant to be made available to other REL modules it must be defined with the ENT opcode, otherwise, other modules wouldn´t know it existed and the linker would not be able to reconcile it.

The following example of a segment of a REL module will illustrate the use of this opcode:

```
                STA  POINTER      ;some meaningless code
                INC  POINTER      ;for our example
                BNE  SWAP         ;CAN BE USED AS NORMAL
                JMP  CONTINUE
     SWAP       EXT               ;MUST BE DEFINED IN THE
                LDA  POINTER      ;CODE PORTION OF THE
                STA  PTR          ;MODULE AND NOT USED
                LDA  POINTER+1    ;AS AN EQUated label
                STA  PTR+1
   * etc.
```

Note that the label SWAP is associated with the code in line 26 and that the label may be used just like any other label in a program. It can be branched to, jumped to, used as a subroutine, etc.

ENT labels will be flagged in the symbol table listing with an "E."

DS (Define Storage)

```
 DS  \
 DS  \expression
        DS \                [ skip to next REL file, fill mem
                              with zeros to next page break ]
        DS \1               [ skip to next REL file, fill mem
                              with the value 1 to next page ]
```

When this opcode is found in an REL file it causes the linker to load the next file in the "linker name file" at the first available page boundary and to fill memory either with zeros or with the value specified by the expression. This opcode should be places at the end of your source file.

ERR (force an ERRor)

```
 ERR  \expression
        ERR  \$4200                   [ error if current code
                                        passes address $4200 ]
```

This opcode will instruct the linker to check that the last byte of the current file does not extend to "expression" or beyond. Note that the expression must be absoulute and not a relative expression.

If the linker finds that this is not the case, linking will abort with the message: CONSTRAINT ERROR:, followed by the value of the expression in the ERR opcode.

Note that the position of this opcode in a REL file has no bearing on its action. It is recommended that it be put at the end of a file.

You can see how this works by trying to link the PI file on the Merlin disk to an address greater than $81C.

LINK (LINK REL files, this is an editor command)

```
 LINK  adrs   "filename"                   [  DOS  3.3 command ]
 LINK  adrs  "pathname"                   [ ProDOS command ]

        LINK $1000 "NAMES"               [ link files in NAMES   ]
        LINK $2000 "/MYPROG/NAMES"       [ link files these ]
```

This editor command invokes the linking loader. For example, suppose you want to link the object files whose names are held in a "linker name file" called NAMES (DOS 3.3 or ProDOS with the prefix set). Suppose the start address desired for the linked program is $1000. Then you would type: LINK $1000 "NAMES" <RETURN>. (The final quote mark in the name is optional and you can use other delimiters such as "´" or ";".) The specified start address has no effect on the space available to the linker.

Note that this command is only accepted if there is no current source file in memory, since the linker would destroy it.

Linker Name Files (DOS 3.3)

The linker name file is just a text file containing the file names of the REL object modules you want linked. It should be written with the Merlin editor and written to the disk with the "W" EXEC command. (Remember to type a space to start the filename for the W command if you don´t want the "T." appended to the start of the name.) Thus if you want to link the object files named MYPROG.START, MYPROG.MID, and LIB.ROUTINE,D2, you would create a text file with these lines:

```
MYPROG.START
MYPROG.MID
LIB.ROUTINE,D2
```

Then you would write this to disk with the "W" command under the filename (for example) MYPROG.NAMES. (Use any filename you wish here, it is not required to call it NAMES.) Then you would link these files with a start address of $1000 by typing NEW and then issueing the editor command: LINK $1000 "MYPROG.NAMES".

The linker will not save the object file it creates. Instead, it sets up the object file pointers for the EXEC mode Object command ("O") and returns you directly to EXEC mode upon the completion of the linking process.

Linker Name Files (ProDOS)

The linker name file is just a specially formatted file (of any type) containing the pathnames of the LNK files you want linked. This file is most easily created by assembling a source file with the proper format, as follows: Each pathname in the source file should be given the form

```
                STR  "pathname",00
```

Be careful to include the 00 at the end. This is vital. The entire source file must end with a BRK (another 00). This tells the linker that there are no more pathnames in the file. Thus if you want to link the LNK files names /MYDISK/START, /MYDISK/MID, AND /OTHERDISK/END you would make a source file containing these lines:

```
                STR  "/MYDISK/START/,00
                STR  "/MYDISK/MID",00
                STR  "/OTHERDISK/END",00
                BRK
```

It is best to use full pathnames as shown, but this is not required. You should then assemble this file and save the object code as, for example, /MY DISK/MYPROG/NAMES. (Use any pathname you want here, it is not necessary to have NAMES in a subdirectory nor to call it NAMES.) Then you can link these files to address $803 by typing NEW and then: LINK $803 "/MYDISK/MYPROG/NAMES" <RETURN> in the editor.

The file type used by the object save command is always the file type used in the last assembly. Thus it is BIN unless the last assembly had a TYP opcode and then it will be that type. This then will be used by the object save command after you link a group of files. (that is, the linker does not change this type.) If you make a mistake and the file gets saved under a type you did not want, just delete the file, change the type by going to the monitor and changing location $BE52 to the correct type, return and resave the object code. You could also just assemble an empty file, which would reset the object type to BIN ($06) but this would defeat the object save command and you would have to link the files again.

The Linking Process (DOS 3.3 and ProDOS)

Various error messages may be sent during the linking process (see the ERRORS section of this manual for more information). If a DOS error occurs involving the file loading, then that error message will be seen and linking will abort. If the DOS error FILE TYPE MISMATCH occurs after the message "Externals:" has been printed then it is being sent by the linker and means that the file structure of one of the files is incorrect and the linking cannot be done.

The messsage PROGRAM TOO LARGE may occur for two reasons. Either the object program is too large to accept (the total object size of the linked file cannot exceed about $A100) or the linking dictionary has exceeded its allotted space ($B000 long). Each of these possibilities is exceedingly remote.

After all files have been loaded, the externals will be resolved. Each external label referenced will be printed to the screen and will be indicated to have been resolved or not resolved. An indication is also given if an external reference corresponds to duplicate entry symbols. With both of these errors the address of the field (one or two bytes) effected is printed. This is the address the field will have when the final code is BLOADed.

This listing may be stopped at any point using the space bar. The space bar may also be used to single step through the list. If you press the space bar while the files are loading then the linker will pause right after resolving the first external reference.

The list can be sent to a printer by using the PRTR or PR# commands prior to the LINK command. At the end, the total number of errors (external references not resolved and references to duplicate entry symbols) will be printed. After hitting a key you will be sent to EXEC mode and can save the linked object file with the object save command, using any filename (or pathname) you please. You can also return to the editor and use the GET command to move the linked code to main memory.

TECHNICAL INFORMATION

The source is placed at STARTOFSOURCE when loaded, regardless of its original address.

The important pointers are:

START OF SOURCE	in	$A,$B	(set to $901 unless changed)
HIMEM	in	$C,$D	(defaults to $9853 in DOS 3.3 defaults to $AA00 in ProDOS)
END OF SOURCE	in	$E,$F	

Note that HIMEM does not change unless a USER routine or utility program changes locations $73, $74. Such a change will be copied automatically into locations $C, $D.

General Information (DOS 3.3 only)

When you exit to BASIC or to the monitor, these pointers are saved on the RAM card at $E00A-$E00F. They are restored up on re-entry to MERLIN.

Entry into MERLIN replaces the current I/O hooks with the standard ones and reconnects DOS. This is the same as typing PR#0 and IN#0 from the keyboard. Entry to the EDITOR disconnects DOS, so that you can use labels such as INIT without disastrous consequences. Re-entry to EXEC MODE disconnects any I/O hooks that you may have established via the editor's PR# command, and reconnects DOS. Exit from assembly (completion of assembly or CTRL-C) also disconnects I/O hooks.

Re-entry after exit to BASIC is made by the "ASSEM" command. Simply use "ASSEM" wherever a DOS command is valid (for example, at the BASIC prompt). A BRUN MERLIN or a disk boot will also provide a warm re-entry and will not reload MERLIN if it is already there. A reload may be forced by typing BRUN BOOT ASM which would then be a cold entry, "destroying" any file in memory.

General Information (ProDOS and DOS 3.3)

If during assembly the object code exceeds usable ram then the code will not be written to memory, but assembly will appear to proceed as normal and its output sent to the screen or printer. The only clue that this has happened, if not intentional, is that the OBJECT CODE SAVE command at EXEC level is disabled in this event. There is ordinarily a 16K space for object code, which can be changed with the OBJ opcode.

Symbol Table

The symbol table is printed after assembly unless LST OFF has been invoked. It is displayed first sorted alphabetically and then sorted numerically. The symbol table can be aborted at any time by pressing CTRL-C. Stopping it in this manner will have no ill effect on the object code which was generated. The symbol table is flagged as follows:

MD	=	Macro Definition
M	=	Label defined within a Macro
V	=	Variable (symbols starting with "]")
?	=	A symbol that was defined but never referenced
X	=	External symbol
E	=	Entry symbol

local labels are not shown in the symbol table listing.

When in EDIT mode, MERLIN takes total control of input and output. The effect of typing a control character will be as described in this manual and NOT as described in the manual for your 80 column card. For example, CTRL-L will not blank the screen, but is the case toggle. CTRL-A, which acts as a case toggle on many 80 column cards, will not do this in EDIT mode and simply produces a CTRL-A in the file line.

Ultraterm Information

When in the editor the ULTRATERM mode can be altered by the ESCAPE sequence given in the ULTRATERM manual. Thus, the following commands give the indicated effects:

ESC 0	40 x 24	(same effect as VID $10 or 16)
ESC 1	80 x 24	standard character set
ESC 2	96 x 24	
ESC 3	160 x 24	
ESC 4	80 x 24	high quality character set
ESC 5	80 x 32	
ESC 6	80 x 48	
ESC 7	132 x 24	
ESC 8	128 x 32	

Exit to EXEC mode will return to the default state as set up in the HELLO program for DOS 3.3 or the PARMS file for ProDOS and the same is true of a VID 3 command.

Except for the normal 24 x 80 format, support for the ULTRATERM depends on the card being in slot 3.

There may be problems if you try to send things to the printer while in some of the ULTRATERM modes. It is recommended that you switch to 40 columns before doing this. "CONTROL-I 80N" in the PRTR command sometimes overcomes the problem.

Memory Allocation with Merlin

The memory areas $300-$3EF in main memory and $800-$FFF in auxiliary memory are available for user supplied USER and USR routines. The page three area in main memory is intended for I/O interface routines. (One cannot send a character to COUT, for example, from auxiliary memory.) Merlin does not use these areas. Zero page locations $90-$9F are not used by Merlin and are reserved for USER routines (note that the XREF program uses these locations). Zero page locations $60-$6F are reserved for user supplied routines and may be used as you wish. No other zero page locations are available.

Configuration (ProDOS version)

Configuration data is kept in a file called PARMS which is loaded when the assembler is run. To change the data just change the source file PARMS.S and reassemble it.

Configuration (DOS 3.3 version)

The DATA statements in the Applesoft boot program "HELLO" contain the configuation information. To change the data just LOAD HELLO, change the data in the DATA statements and SAVE HELLO.

Description of data for both DOS 3.3 and ProDOS configuations:

DATA #	DEFAULT	PURPOSE
1	60	Number of lines per page (for PRTR)
2	0	Lines to skip at page perforation (0 sends a form feed character
3	80	Number of characters per line (for PRTR)
4	$80	Must be $80 if printer does its own CR at end of line, otherwise should be 0
5	$83	80 column flag. Should be $80+3 if 80 column card is in slot 3 (or Apple 80 col card) is to be selected upon boot. Otherwise 0. MUST BE $83 WITH ProDOS.
6,7	$901	Source file start address, must not be less than $901
8,9	$AA00	SHOULD NOT BE CHANGED
10,11	$901	End of source pointer. Must equal the Source file start address
12	$DE "^"	The editor´s wild card character
13	4	Number of fields per line in symbol table printout.
14	$AF "/"	Character searched for by "UPDATE SOURCE" entry to assembler. If this is 0 the question will be bypassed.
15,16,17	14,20,31	The default tabs for editor and assembler, note that these values are relative to the left side of screen.
18	8	Number of object bytes/line after the first line.

DATA #	DEFAULT	PURPOSE
19	5	Error/bell flag and Ultraterm start parameters. The high bit, if on, will force the assembler to pause forever for a keypress at an error; if off, a sound continues for 20 seconds and then assembly continues. The V bit, if set disables some bells. The low nibble determines the default mode of the Ultraterm if you are using that. The value 5 or $85 gives the 32X80 mode.
20	$40	Cursor flag. Gives regular cursor if this is $40 and block cursor if 0. The Apple 80-col card must have te block cursor and this flag will be overridden if you are using that card.
21	0	LSTDO default: 0,1=LSTDO ON, >1=LSTDO OFF. Bit 0, if clear, causes shift to 40 columns when a PRTR command is issued.
22	72	Column at which the cycle count will be printed when using the CYC opcode.
23	$EC	Cursor type for Ultraterm. Must be changed if the Ultraterm mode is changed (see byte 19)
24-44	"$F1" to "$F7"	File type names for the user defined file types $F1 through $F7. These names will be shown in the directory when cataloged by Merlin. ProDOS ONLY.

64K Merlin and Merlin Pro Source Files

Source files from the 64k Merlin can be loaded directly into DOS 3.3 Merlin Pro. To use 64k Merlin source files with ProDOS Merlin Pro you must use the CONVERT utility supplied with the ProDOS User´s Disk. Some changes may be required to the source due to some of the missing pseudo opcodes in Merlin Pro. If your program uses HIMEM: or SYM, they should be deleted. If your program uses the ERR opcode to check whether SYM or HIMEM: have been set, they should be deleted. If your program uses Sweet 16 then the enabling opcode SW will have to be inserted. Also, any OBJ opcodes will have to be removed since the meaning of this opcode has been changed.

ProDOS Merlin Pro Notes

The ProDOS version uses TXT files exclusively for source files. This includes files intended for the PUT or USE opcodes, and all such files must have the ".S" extension in the file name (which is provided by the assembler for all loads and saves). It is suggested that you keep files intended for PUT or USE in a subdirectory. For example you could save a file named MYPUT under the pathname LIB/MYPUT. It would then be called in an assembly program by: PUT LIB/MYPUT, or PUT /PREFIX/LIB/MYPUT if it is in the volume called PREFIX.

If you save a file under a directory name that does not exist, a subdirectory will be created under that name. For example, suppose you want to save your current source SRC in the volume MYVOL and in the subdirectory SUB which does not exist in the MYVOL directory. Then merely type /MYVOL/SUB/SRC when the pathname is requested (or just SUB/SRC if /MYVOL/ is the prefix) and the subdirectory SUB will be automatically created and the file SRC placed in it.

It is wise to use a full pathname in operands of the SAV, USES and PUT opcodes, since otherwise the current prefix will be attached to the name and that may not be the prefix you want.

Slot and drive parameters are NOT acceptable by any commands or opcodes. You MUST use pathnames.

Since the ProDOS version of Merlin runs under its own interpreter rather than the BASIC interpreter, there is no warm re-entry as with the DOS 3.3 version.

There is no equivalent of the BASIC CAT or CATALOG commands as "disk commands." The interpreter automatically selects the catalog format for the "C" command according to whether you are in 40 or 80 column mode.

The ProDOS volume /RAM/ is disconnected by Merlin Pro since it uses all of auxiliary memory.

Transferring Source Files from DOS 3.3 TO ProDOS Merlin Pro

There are two methods of transferring files from the DOS 3.3 versions of Merlin to the ProDOS version. Since the ProDOS version uses text files only, you could load files into the DOS 3.3 version and write them as text files and then transfer them with Apple´s CONVERT program. Unfortunately, CONVERT is not a literal transfer, as it will clear the high bits in the file. The ProDOS version of Merlin will set the High bits again, but the tabbing in the editor will be fouled up by this procedure. However, you merely have to type FIX in the editor and resave the source to remedy this problem. Files intended for "PUT" or "USE" should be resave because, otherwise, assembly will be slowed.

Another method is to transfer the files as binary files from DOS 3.3 and use the fact that the ProDOS version of Merlin has the ability to load binary files (or any type). (This does NOT apply to saving.) After loading a binary source file, it should be deleted and saved back (as a TXT file). The Load command automatically permits loading of TXT of BIN files. Other types of files can be loaded by changing the byte used to designate source file type which is kept in location $BE5D (this ordinarily holds a 4).

Since the ProDOS version of the assembler does not use the "T." syntax of the DOS 3.3 version for PUT files, there will be some renaming of such files that will be necessary.

ERROR MESSAGES

BAD OPCODE

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

BAD ADDRESS MODE

The addressing mode is not a valid 6502 instruction; for example, JSR (LABEL) or LDX (LABEL),Y.

BAD BRANCH

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

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

DUPLICATE SYMBOL

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

MEMORY FULL

This is usually caused by one of two conditions: Source code too large or symbol table too large. See "Special Note" at the end of this section.

UNKNOWN LABEL

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

NOT MACRO

Forward reference to a MACRO, or reference by PMC or >>> to a label that is not a MACRO.

NESTING ERROR

Macros nested more than 15 deep or conditionals nested more than 8 deep will generate this error.

BAD "PUT"

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

BAD "SAV"

This is caused by a SAV inside a macro or a SAV after a multiple OBJ after the last SAV.

BAD INPUT

This results from either no input ([RETURN] alone) or an input exceeding 37 characters in answer to the KBD opcode´s request for the value of a label.

BREAK

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

BAD LABEL

This is caused by an unlabeled EQU, MAC, ENT or EXT, a label that is too long (greater than 13 characters) or one containing illegal characters (a label must begin with a character at least as large in ASCII value as the colon and may not contain any characters less than the digit zero).

BAD ORG

Results from an ORG at the start of a REL file.

BAD OBJ

An OBJ after code start or OBJ not within $4000 to $BFE0.

BAD REL

A REL opcode occurs after some labels have been defined.

BAD EXTERNAL

EXT or ENT in a macro or an equate of a label to an expression containing an external, or a branch to an external (use JMP).

BAD VARIABLE

This occurs when you do not pass the number of variables to a macro that the macro expects. It can also occur for a syntax error in a string passed to a macro variable, such as a literal without the final quote.

ILLEGAL FORWARD REFERENCE

A label equated to a zero page address after it has been used. This also occurs when an unknown (on the first pass) label is used for some things that must be able to calculate the value on the first pass (e.g. ORG< OBJ DUM). It also occurs if a label is used before it is defined in a DUM section on zero page

TWO EXTERNALS

Two or more externals in an operand expression.

DICTIONARY FULL

Overflow of the relocation dictionary in a REL file.

256 EXTERNALS

The file has more than 255 externals.

ILLEGAL RELATIVE ADRS

In REL mode a multiplication, division or logical operation occurs in a relative expression. This also occurs for an operand of the type #>expr or a DFB >expr when the expr contains an external and the offset of the value of the expr from that of the external exceeds 7.

ILLEGAL CHAR IN OPERAND

A non-math character occurs in the operand where the assembler is expecting a math operator. This usually occurs in macro calls with improper syntax resulting from the textual substitution.

ILLEGAL FILE TYPE (ProDOS version only)

TYP opcode used with an illegal operand. The operand must evaluate to 0,6,F0-F7, or FF.

GENERAL NOTE: When an error occurs that aborts assembly, the line containing the error is printed to the screen. This may not have the same form as it has in the source, since it shows any textual substitutions that may have occcurred because of macro expansion. If it is in a macro call, the line number will be that of the call line and not of the line in the macro (which is unknown to the assembler).

Special Note - MEMORY FULL Errors

There are three common causes for the MEMORY FULL error message. A more detailed description of this problem and some ways to overcome it follow.

MEMORY FULL IN LINE: xx. Generated during assembly. CAUSE: Too many symbols have been placed into the symbol table, causing it to exceed available space. REMEDY: Make the symbol table larger by setting OBJ to $BFE0 and use DSK to assemble directly to disk.

ERR:MEMORY FULL. Generated immediately after you type in one line too many. CAUSE: The source code is too large and has exceeded available ram. REMEDY: Break the source file up into smaller sections and bring them in when necessary by using the "PUT" pseudo-op.

ERROR MESSAGE: None, but no object code will be generated (there will be no OBJECT information displayed on the EXEC menu). CAUSE: Object code generated from an assembly would have exceeded the avaiable 16K space. REMEDY: Set OBJ to an address less than its $8000 default or use DSK.

SOURCEROR

Introduction

SOURCEROR is a sophisticated and easy to use co-resident disassembler designed to create MERLIN source files out of binary programs, usually in a matter of minutes. SOURCEROR disassembles SWEET 16 code as well as 6502, 65C02 and 65802 code.

Using SOURCEROR

1. [DOS 3.3] From the EXEC mode, type C to CATALOG Merlin Pro . At the Command prompt, type BRUN SOURCEROR.

   [ProDOS] From the EXEC mode, type D for DISK COMMAND. At the prompt, type BRUN/MERLIN/SOURCEROR/OBJ (assuming this is the appropriate prefix etc.).

2. From the EDIT mode, use ESC CTRL-Q (not Escape-4) to set the screen to 40 columns, then type USER. If the screen is in 80 columns, the USER command will be ignored.

3. You will be asked if you want to load an object file to be disassembled. If you have already loaded the object file prior to using SOURCEROR, type N and skip to step 5. If yes, type Y and enter the filename. It will be loaded showing the load address and end of program address.

   Note: If you type CTRL-S after the filename to be loaded, files using a RAM version of SWEET 16 can be disassembled.

4. Next, you will be asked to press RETURN if the program to be disassembled is at its original (running) location, or you must specify in hex the present location of the file to be disassembled. You will then be asked to give the ORIGINAL location of that program.

5. Finally, the screen displays the commands available for disassembly. You may begin disassembling now, or use any of the other commands shown. Your first command MUST include a hex address. Thereafter this is optional, as explained later.

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

6. When you are all done using SOURCEROR, you should type USER1 from the EDITOR to get rid of SOURCEROR and free up the memory used by the disassembler.

Commands Used in Disassembly

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

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

More commonly, you will specify an address less than the present default value. In this case, the disassembler checks to see if this address equals the address of one of the previous lines. If so, it simply backs up to that point. If not, then it backs up to the next used address and creates a new ORG. Subsequent source lines are "erased". It is generally best to avoid new ORGs when possible. If you get a new ORG and don't want it, try backing up a bit more until you no longer get a new ORG upon disassembly.

This "backup" feature allows you to repeat a disassembly if you have, for example, used a HEX or other command, and then change your mind.

Command Descriptions

L (List)

This is the main disassembly command. It disassembles 20 lines of code. It may be repeated (e.g. 2000LLL will disassemble 60 lines of code starting at $2000). If a JSR to the SWEET 16 interpreter is found, disassembly is automatically switched to the SWEET 16 mode.

Command L always continues the present mode of disassembly (SWEET 16 or normal).

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

S (SWEET)

This is similar to L, but forces the disassembly to start in SWEET 16 mode. SWEET 16 mode returns to normal 6502 mode whenever the SWEET 16 RTN opcode is found.

N (Normal)

This is the same as L, but forces disassembly to start in normal 6502 mode.

H (Hex)

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

T (Text)

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

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

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

This, or TT, may also have to be used to establish the correct boundary between a regular ASCII string and a flashing one. It is usually obvious where this should be done.

W (Word)

This disassembles the next two bytes at the current location as a DA opcode. Optionally, if the command WW is used, these bytes are disassembled as a DDB opcode.

If W- is used as the command, the two bytes are disassem bled in the form DA LABEL-1. The latter is often the appropriate form when the program uses the address by pushing it on the stack. You may detect this while disassembling, or after the program has been disassembled. In the latter case, it may be to your advantage to do the disassembly again with some notes in hand.

Housekeeping Commands

/ (Cancel)

This essentially cancels the last command. More exactly, it re-establishes the last default address (the address used for a command not necessarily attached to an address). This is a useful convenience which allows you to ignore the typing of an address when a backup is desired.

As an example, suppose you type T to disassemble some text. You may not know what to expect following the text, so you can just type to L to look at it. Then if the text turns out to be followed by some Hex data (such as $8D for a carriage return), simply type / to cancel the L and type the appropriate H command.

R (Read)

This allows you to look at memory in a format that makes imbedded text stand out. To look at the data from $1000 to $10FF type 1000R. After that, R alone will bring up the next page of memory. The numbers you use for this command are totally independent of the disassembly address.

However, you may disassemble, then use (address)R, then L alone, and the disassembly will proceed just as if you never used R at all. If you don′t intend to use the default address when you return to disassembly, it may be wise to make a note on where you wanted to resume, or to use the / command before the R command.

Q (Quit)

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

```
2341-  4C  03  E0        JMP  $E003
2344-  A9  BE  94        LDA  $94BE,Y
```

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

Final Processing

After the Q command, the program does some last minute processing of the assembled code. If you hit RESET at this time, you will return to MERLIN and lose the disassembled code.

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

When the processing is done, you are returned to Merlin with the newly created source in the text buffer. You can use Merlin's Save command to save it to disk when you want.

Dealing with the Finished Source

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

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

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

NOTE:If you try to assemble the source under these conditions, you will get an error as soon as the equates appear. If, as eventually you should, you move the equates to the start of the program, you will not get an error, but the assembly MAY NOT BE CORRECT.

It is important to deal with this situation first as trouble could occur if, for example, the disassembler finds the data AD008D. It will disassemble it correctly, as LDA $008D. The assembler always assembles this code as a zero page instruction, giving the two bytes A5 8D. Occasionally you will find a program that uses this form for a zero page instruction. In that case, you will have to insert a character after the LDA opcode to have it assemble identically to its original form. Often it was data in the first place rather than code, and must be dealt with to get a correct assembly.

The Memory Full Message

When the source file reaches within $600 bytes of the end of its available space you will see MEMORY FULL and "HIT A KEY". When you hit a key, SOURCEROR will go directly to the final processing. The reason for the $600 byte gap is that SOURCEROR needs a certain amount of space for this processing. There is a "secret" override provision at the memory full point. If the key you hit is CTRL-O (for override), then SOURCEROR will return for another command. You can use this to specify the desired ending point. You can also use it to go a little further than SOURCEROR wants you to, and disassemble a few more lines. Obviously, you should not carry this to extremes. If you get too close to the end of available space, Sourceror will no longer accept this overide and will automatically start the final processing.

Changing Sourceror´s Label Tables

The label tables used by Sourceror are just assembled Merlin source files. The source file is on the Merlin disk and can be modified directly by the user. It must be assembled and saved under the same name as the previous label file, i.e. you have to replace the old existing file.

APPLESOFT LISTING INFORMATION

SOURCEROR.FP

A fully labelled and commented source listing of Applesoft BASIC can be generated by the program SOURCEROR.FP on the opposite side of the ProDOS MERLIN diskette.

This program works by scanning the resident copy of Applesoft present in your computer and generating text files containing the bulk of Applesoft BASIC: APSOFT.1, APSOFT.2, APSOFT.3, AND APSOFT.4.

To conserve space, these files contain macros that are defined in another file on the disk entitled, APPLESOFT.S. This file, when assembled using the PRTR command, will print out a nicely formatted disassembly of Applesoft, automatically bringing in and using the APSOFT files as necessary. Exact details on doing this are outlined below.

PLEASE NOTE that this is NOT an "official" source listing from Apple Computer, Inc., but rather a product of the Author's own research and interpretation of the original Applesoft ROM. Apple Computer, Inc. was not in any way involved in the preparation of this data, nor was the final product reviewed for accuracy by that company. Use of the term APPLE should not be construed to represent any endorsement, official or otherwise, by Apple Computer, Inc.

Additionally, Roger Wagner Publishing makes no warranties concerning the accuracy or usability of this data. It is provided solely for the entertainment of users of the MERLIN assembler.

WARNING: SOURCEROR.FP and some temporary work files are DELETED when SOURCEROR.FP is BRUN. For this reason, you should make a backup copy of the SOURCEROR.FP side of the MERLIN disk with the COPYA program on the DOS 3.3 System Master diskette. Use the backup copy to make the Applesoft listing as explained next.

Steps to print the Applesoft Disassembly

1. Boot ProDOS Merlin,

2. BRUN SOURCEROR.FP from Merlin´s Disk command, use your backup copy of the SOURCEROR.FP disk (see warning above).

3. When SOURCEROR.FP finishes, L)oad the file APPLESOFT.

4. Type the following, to print the listing on your printer:

```
PRTR 1 "I80N" APPLESOFT LISTING
ASM
```

In the example above, the PRTR command will send output to slot 1, initialize the printer interface card with <CTRL I-80N" (the I is in inverse), and will print "APPLESOFT LISTING" as a header at the top of every page.

MERLIN will then ask "GIVE VALUE FOR SAVEOBJ :" This refers to whether or not you want to save object code generated by the assembly. It is recommended that you answer, "0". This is all you need to do to begin the printing process. If you answer "1", you will save object code at the cost of slowing down the system. Saved object code allows you to verify it against where it was taken from.

MERLIN will now execute the first assembler pass. The disk will be accessed a few times, sometimes with long periods between accesses. This is normal. The entire first pass takes about 3.5 minutes.

MERLIN will then begin to print out a completely disassembled and commented listing of Applesoft. It will take 105 pages (including the symbol tables) and nearly an hour and a half to print out (at a printer rate of 80 characters per second).

Applesoft Source Cross Reference Listing

Although 105 pages of Applesoft source would seem like enough to keep one busy for at least a year, Merlin also offers another source of Applesoft internal information - Applesoft internal address, subroutine and zero page cross references. By using the XREFA utility with the Applesoft source you can produce a listing of every subroutine, zero page address and where they are used and called. This is invaluable information for the programmer who desires to make use of the routines inside Applesoft in his own programs.

Assume, for example, that a user program is called by a running Applesoft program. Also assume that the programmer makes calls to some internal Applesoft routines and that the programmer wishes to use zero page locations $50 and $51 as temporary registers or pointers. This cross reference will immediately inform the programmer whether or not the routines that his program use will destroy the contents of these two locations and cause difficult to find bugs in his program.

Steps to print an Applesoft cross reference:

1. Load the APPLESOFT file from the SOURCEROR.FP disk,

2. Quit to the EXEC mode and press D for disk command,

3. BRUN /MERLIN/UTIL/XREFA,

4. Go to the Editor with the E command,

5. Issue the PRTR command: PRTR 1 "I80N" APPLESOFT XREF

6. Issue the following command: USER 3

7. Then ASM to begin the assembly.

When this is done, the Applesoft source will again be assembled. This time, however, the XREFA program will limit your printed output to the cross reference table. Note that this process also takes quite a bit of time prior to printing.

GLOSSARY

ABORT	-terminate an operation prematurely.
ACCESS	-locate or retrieve data.
ADDRESS	-a specific location in memory.
ALGORITHM	-a method of solving a specific problem.
ALLOCATE	-set aside or reserve space.
ASCII	-industry standard system of 128 computer codes assigned to specified alpha-numeric and special characters.
BASE	-in number systems, the exponent at which the system repeats itself; the number of symbols required by that number system.
BINARY	-the base two number system, composed solely of the numbers zero and one.
BIT	-one unit of binary data, either a zero or a one.
BRANCH	-continue execution at a new location.
BUFFER	-large temporary data storage area.
BYTE	-Hex representation of eight binary bits.
CARRY	-flag in the 6502 status register.
CHIP	-tiny piece of silicon or germanium containing many integrated circuits.
CODE	-slang for data or machine language instructions.
CTRL	-abbreviation for control or control character.

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

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

DECREMENT -decrease value in constant steps.

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

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

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

EQUATE -establish a variable.

EXPRESSION -actual, implied or symbolic data.

FETCH -retrieve or get.

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

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

HEX -the Hexadecimal (BASE 16) number system, composed of the numbers 0-9 and the letters A-F.

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

HOOK -vector address to an I/O routine or port.

INCREMENT -increase value in constant steps.

INITIALIZE -set all program parameters to zero, normal, or default condition.

I/O -input/output.

INTERFACE -method of interconnecting peripheral equipment.

INVERT -change to the opposite state.

LABEL -name applied to a variable or address, usually descriptive of its purpose.

LOOKUP -slang; see table.

LOW-ORDER -the second, or least significant byte of a two-byte Hex address or value.

LSB -least significant (bit or byte) one with the least value.

MACRO -in assemblers, the capability to "call" a code segment by a symbolic name and place it in the object file.

MICROPROCESSOR -heart of a microcomputer. (In the Apple, the 6502 chip).

MOD -algorithm returning the remainder of a division operation.

MODE -particular sub-type of operation.

MODULE -portion of a program devoted to a specific function.

MNEMONIC -symbolic abbreviation using characters helpful in recalling a function.

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

NULL -without value.

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

OFFSET -value of a displacement.

OPCODE -instruction to be executed by the 6502.

OPERAND -data to be operated on by a 6502 instruction.

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

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

PERIPHERAL -external device.

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

PORT -physical interconnection point to peripheral equipment.

PROMPT -a character asking the user to input data.

PSEUDO -artificial, a substitute for.

RAM -Random Access Memory.

REGISTER -single 6502 or memory location.

RELATIVE -branch made using an offset or displacement.

ROM -Read Only Memory.

SIGN BIT -bit eight of a byte; negative if value greater than $80.

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

STACK -temporary storage area in RAM used by the 6502 and assembly language programs.

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

SWEET 16 -program which simulates a 16 bit microprocessor.

SYMBOL	-symbolic or mnemonic label.
SYNTAX	-prescribed method of data entry.
TABLE	-list of values, words, data referenced by a program.
TOGGLE	-switch from one state to the other.
VARIABLE	-alpha-numeric expression which may assume or be assigned a number of values.
VECTOR	-address to be referenced or branched to.

UTILITIES

Formatter

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

To use FORMATTER, you should first BRUN it from EXEC mode. FORMATTER will then load itself into high memory.

This will simply set up the editor´s USER vector. To format a file which is in memory, issue the USER command from the editor.

The formatter program will request a range to format. If you just specify one number, the file will be formatted from that line to the end. Then you will be asked for a line length, which must be less than 250. Finally, you may specify whether you want the file justified on both sides (rather than just on the left).

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

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

Don´t forget to use the TABS command before printing out a formatted file.

XREF, XREFA

These utilities provide a convenient means of generating a cross-reference listing of all labels used within a Merlin assembly language (i.e., source) program.

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

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

This is where the XREF programs come in.

XREF gives you a complete alphabetical and numerical printout of label usage within an assembly language program. XREFA gives a cross reference table by ADDRESS. This is more useful for large sources containing lots of PUT files. It also does not use as much space for its cross-reference data and therefore can handle larger source files than XREF.

XREF.H and XREFA.H are ProDOS versions of the XREF and XREFA programs that use a page of high memory rather than page 3 memory. This is intended as a convenience for people who have a clock driver in page 3.

Sample Merlin Symbol Table Printout:

Symbol table - alphabetical order:

```
   ADD      =$F786      BC      =$F7B0      BK      =$F706
```

Symbol table - numerical order:

```
   BK       =$F706      ADD     = $F786     BC      =$F7B0
```

Sample Merlin XREF Printout:

```
Cross referenced symbol table - alphabetical order:

     ADD      =$F786  101    185*
     BC       =$F7B0  90     207*
     BK       =$F706  104    121*

Cross referenced symbol table - numerical order:

     BK       =$F706  104     121*
     ADD      =$F786  101     185*
     BC       =$F7B0  90      207*
```

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

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

XREF Instructions

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

2. C)atalog the disk and when Merlin asks you for a COMMAND: after the Catalog, enter: BRUN XREF.

2a. For ProDOS Merlin, press D)isk and when Merlin asks for a COMMAND: enter: BRUN /MERLIN/UTIL/XREF.

3. Enter the Editor, then type the appropriate USER command:

   USER 0 -Print assembly listing and alphabetical cross reference only. (USER has the same effect as USER 0).

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

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

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

USER commands 0-3 (above) cause labels within conditional assembly areas with the DO condition OFF to be ignored and not printed in the cross reference table.

There are additional USER commands (4-7) that function the same as USER 0-3, except that they cause labels within conditional assembly areas to be printed no matter what the state of the DO setting is. The only exception to this is that labels defined in such areas and not elsewhere will be ignored.

NOTE: You may change the USER command as many times as you wish (e.g., from USER 1 to USER 2). The change is not permanent until you enter the ASM command (below).

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

Since the XREF programs require assembler output, code in areas with LST OFF will not be processed and labels in those areas will not appear in the table. In particular, it is essential to the proper working of XREF that the LST condition be ON at the end of assembly (since the program also intercepts the regular symbol table output). For the same reason, the CTRL D flush command must not be used during assembly. The program attempts to determine when the assembler is sending it an error message on the first pass and it aborts assembly in this case, but this is not 100% reliable.

Another thing to look out for when using macros with XREF. Labels defined within macro definitions have no global meaning and are therefore not cross-referenced.

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

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

XREFA

This is an ADDRESS cross reference program and is handy when you have lots of PUT files. Since this program needs only four bytes per cross reference instead of six, it can handle considerably larger sources. Also the "where defined" reference is not given here because it would equal the value of the label except for EQUated labels where it would just indicate the address counter when the equate is done. This also saves considerable space in the table for a larger source.

PRINTFILER

PRINTFILER is a utility included on the Merlin diskette that saves an assembled listing to disk as a sequential disk file. It optionally allows you to also select "file packing" for smaller space requirements and allows you to turn video output off for faster operation.

Text files generated by PRINTFILER include the object code portion of a disassembled listing, something not normally available when saving a source file. This allows a complete display of an assembly language program and provides the convenience of not having to assemble the program to see what the object code looks like.

Applications

Applications include:

-Incorporating the assembled text file in a document being prepared by a word processor.

-Sending the file over a telephone line using a modem.

-Mailing the file to someone who wants to work with the complete disassembly without having to assemble the program (such as magazine editors, etc.)

How To Use PRINTFILER from DOS 3.3

1. From EXEC mode, make sure that you´ve Saved any source file that you may be working on (select the Drive to save it on, first), select the Drive containing PRINTFILER (usually this is on the Merlin disk) and do a Catalog. When you see the "COMMAND:" prompt, enter BRUN PRINTFILER (You may skip this step if you´ve already BRUN´ed it).

2. Press RETURN, select the Drive containing the file you want to assemble and Load the file into memory. (You may skip this step if you´ve already BRUN PRINTFILER).

3. Quit the editor, select the Drive that you want to save the assembly to, enter the Editor again and enter: USER "your file name" (include the quotes). You may also use the PRTR command if you wish page headers to be sent with your listing. In this case enter the following instead of the USER command: PRTR 8 "filename" page header (note the quotes only for filename).

4. Enter: ASM and after asking whether you want to "UPDATE SOURCE", PRINTFILER will automatically assemble the source file directly to disk. Note that you will not see any thing on your video screen because PRINTFILER is preconfigured to operate with the video output turned off for faster operation.

How to Use PRINTFILER from ProDOS

1. Be certain, the /MERLIN/UTIL/PRINTFILER file is online. Then press D for disk command, and then enter: BRUN /MERLIN/UTIL/PRINTFILER.

2. Load the file you wish to assemble. When you enter the editor, enter: USER "pathname" (include the quotes with this pathname). You may also use the PRTR command if you wish page headers to be sent with your listing. In this case enter the following instead of the USER command: PRTR 8 "pathname" page header (note quotes only for pathname).

3. Enter: ASM and after asking whether you want to "UPDATE SOURCE", PRINTFILER will assemble the source and send the listing to disk. Note that you will not see any thing on your video screen because PRINTFILER is preconfigured to operate with the video output turned off for faster operation.

Changing PRINTFILER´s Options

PRINTFILER has two options that you may change: file packing and video output ("echoing"). In addition, you can make the change temporary or permanent.

File packing reduces the size of the text file saved to disk by replacing blanks from the source file with a single character with its high bit turned off. A listing of a packed file will display the packed blank characters as an inverse letter. (inverse A=1 blank, inverse B=2 blanks, inverse C=3 blanks, etc.)

Unpacking means restoring the text file to its original appearance. Note that while you cannot ASM (assemble) such a file, you can at least read it.

Video "echoing" means printing on the screen what is sent to the disk. The time it takes to do this can slow PRINTFILER down.

The process of turning off video output makes PRINTFILER run approximately 25% faster. Additional speed can be gained by using packed files.

In addition, unpacked files are nearly twice as large as packed files and nearly three times the size of the original source file.

Changing PRINTFILER options

To Change PRINTFILER options (temporarily)

Get into the Editor, enter "MON" and enter:

```
300:00 00   for packed, video off, or.
300:00 80   for packed, video on, or
300:80 00   for unpacked, video off, or
300:80 80   for unpacked, video on, or
```

(normal values are 300:80 00 (unpacked, video off))

Hit RETURN CTRL-Y RETURN to return to EXEC mode. The values you select will stay in effect until you BRUN PRINTFILER again.

To Change PRINTFILER options (permanently)

1. Load PRINTFILER and ASM it. During assembly, it will ask you the following questions in the steps below:

2. After the UPDATE SOURCE? question, PRINTFILER will ask, "GIVE VALUE FOR FORMAT:". If you hit "0", you will turn the Pack option ON. If you hit "1", you will turn the Pack option OFF.

3. PRINTFILER will then ask, "GIVE VALUE FOR MONITOR". If you hit "0", video output will be turned OFF. If you hit "1", video output will be turned ON. PRINTFILER will then immediately assemble into object code.

4. Quit the editor and save the Object code. Any time you BRUN this object code, it will use the values you put in it in steps 2 and 3 above. Thus, it is possible to use different versions of PRINTFILER instead of setting options.

THE 65802 MICROPROCESSOR

The new 65802 microprocessor chip is an enhancement of the 65C02 which supports 16 bit addressing and several new opcodes and addressing modes.

There is a new status bit, called the emulation bit and named E. If this bit is set then the 65802 is totally compatible with the 6502 and 65C02 but recognizes some new opcodes. If this bit is clear then a few things (such as BRK processing) work somewhat differently from the 6502, and 16 bit addressing is possible.

The emulation bit E is affected by just one opcode XCE which exchanges it with the carry bit C of the status register. Since E is not technically part of the status register, PLP does not change it.

When E=0 (emulation off) there are two bits of the status register that control the 16 bit modes of the processor. One of these bits, bit 5, is the unused status bit on the 6502, and the other, bit 4, is the BRK bit on the 6502. The use of this latter bit is made possible by the change in the way a BRK is handled when E=0.

Bit 5 of the status register is called M and selects 8-bit (M=1) or 16 bit (M=0) memory access (by LDA etc.) and accumulator size.

Bit 4 of the status register is called X and selects 8-bit (X=1) or 16 bit (X=0) index register length (affecting the X and Y registers).

When E=0 and M=0, the accumulator is 16 bits long and is called the C register, with A corresponding to the lower 8 bits and with the upper 8 bits being called B. (One still uses LDA, etc., in 16 bit mode, however.) In 16 bit mode (both E and M zero) an instruction LDA $1000 will load the 8-bit A register from $1000 and then load the B register from address $1001.

The stack register is also 16 bits long when E=0. Thus one can put a stack anywhere in memory.

There is a new register called the direct register D when E=0. This register enhances what is called zero page addressing on the 6502, and that addressing mode is called "direct addressing" on the 65802. When E=0 any "direct address mode" such as LDA $40 operates by adding the byte following the opcode ($40 here) to the contents of the direct register, which then forms the effective address for the instruction. If the direct register contains the value $2010, then this example would be equivalent to LDA $2050 (since $2010+$40=$2050), but would execute faster. For fastest execution, the low byte of the direct register D should be kept 0, since extra clock cycles are needed when it is not zero. The effect of this direct register enhancement is to enable the implementation of a "zero page" anywhere in memory. For example, if you place the pointer $1000 in location $300 (0 in $300 and $10 in $301) and if you load the D register with the value $300, then the instruction LDA (0),Y will load the accumulator with the data in memory address $1000+Y (i.e., from the address held in location $300+0 plus the value of Y). Note that if the direct register contains the value zero, then direct addressing is completely equivalent to the old "zero page addressing" in all modes.

"DIRECT" ADDRESSING MODES:

The use of the term "direct" results in such abominations of syntax as "preindexed direct indirect addressing". The more suspicious among us might feel that this terminology was expressly chosen to confuse the beginner and to keep an air of mystery surrounding assembly language programming. We will avoid this terminology and, instead, describe the various addressing modes by simply showing the corresponding assembler operand syntax. We use a single dash "-" to indicate a zero page expression and two dashes "--" to indicate a 16 bit address or value.

Address mode: -

This is the simplest of the "direct" addressing modes. The direct register is added to - (the second byte of the instruction). This forms the effective address. Thus if D=$1234 then LDA $56 will load the accumulator from location $1234+$56=$128A (and from $128B if M=0).

Address mode: -,X

The direct register is added to - and the result is added to the X register to form the effective address.

Address mode: -,Y

This mode is available only for LDX -,Y and STX -,Y. The direct register is added to - and this is added to Y to form the effective address. (Note that the assembler will accept things like LDA $10,Y but since direct,Y mode is not supported by LDA, this is assembled as if it were LDA $0010,Y.) To force the non-direct mode for a LDX or STX instruction, you should use the LDX: or STX: syntax.

Address mode: (-),Y

The direct register is added to - to form an address adrsl. Then the CONTENTS of adrsl,adrsl+1 form an address adrs2. This in turn is added to the Y register to form the effective address. For example, if D=$1234 and locations $125A,$125B contain the address $DBA1 then the instruction LDA ($26),Y will load the accumulator with the byte(s) at $DBA1+Y (and $DBA2+Y if M=0).

Address mode: (-,X)

The direct register is added to - to form an address adrsl. This is added to the X register to form adrs2, Then the CONTENTS of adrs2 and adrs2+1 form adrs3 which is the effective address.

Address mode: (-)

The direct register is added to - to form adrsl. Then the contents of adrsl,adrsl+1 form the effective address.

ABSOLUTE ADDRESSING MODES:

The addressing modes "--", "--,X", and "--,Y" operate in exactly the same manner as on the 6502, except of course for the effect on them of 16 bit mode. (If M=0 then STA $1000 will store A in $1000 and B in $1001.) Thus these need not be detailed here.

IMMEDIATE ADDRESSING:

Immediate addressing refers to things like LDA #3 and CPX #$45. The 65802 also allows these same immediate opcodes to operate on 16 bit data. Whether a particular opcode, such as LDA #, will operate on 8 or 16 bits depends on the values of the status bits M and X. Codes such as LDA #, CMP #, BIT #, (in fact any immediate code not involving the X or Y registers) operate on 16 bit data if the M status bit is clear (M=0). The immediate codes, such as LDX # or CPY #, which involve X or Y operate on 16 bit data if the X status bit is clear. Unfortunately, the assembler (any assembler) has no way of knowing what the state of the M and X status bits will be in a running program. Thus, for the assembler to properly assemble an immediate opcode, it must be informed of the state of these bits. This is done through the MX pseudo-opcode. (The assembler instruction MX %01 tells the assembler that M=0 and X=1, for example. Note the use of binary data here.)

STACK RELATIVE ADDRESSING:

This is a new addressing mode which has no counterpart for the 6502. It comes in both plain and indirect indexed versions. Both of these are supported for the instructions ORA, AND, EOR, ADC, STA, LDA, CMP, and SBC:

-,S

In this mode, the stack register is added to - to form the effective address (which is an address in the stack).

(-,S),Y

In this mode, the stack register is added to - to form an address adrsl. Then the contents of adrsl,adrsl+1 form an address adrs2. The effective address is then adrs2+Y. The purpose of this addressing mode is to use the stack to pass data addresses to subroutines.

INDIRECT LONG ADDRESSING:

This is a new addressing mode, which comes in plain and indexed versions. Both of these are supported by the instructions ORA, AND, EOR, ADC, STA, LDA, CMP, and SBC. Although the assembler supports these addressing modes, they really do nothing useful on the 65802. (They are intended for the extended addressing capabilties of the 65816 chip.):

[-]

The direct register is added to - to form an address adrsl. The contents of adrsl,adrsl+1 is then the effective address. In 16 bit mode the byte at adrsl+2 gives the "bank address".

[-],Y

The direct register is added to - to form an address adrsl. The contents of adrsl,adrsl+1 (and adrsl+2 in 16 bit mode) are then added to the Y register to form the effective address.

BLOCK MOVE ADDRESSING:

This applies only to the two block move opcodes MVP (move forward) and MVN (move backward). See the description of these codes.

NEW OPCODES:

The 65802 supports all the 65C02 instructions (not including the so-called Rockwell codes). In addition it has the following new opcodes:

New PUSH and PULL Instructions

PEA -- (Push effective absolute address)
($F4 - 3 bytes)

This pushes the 16 bit address -- on the stack, high byte first.

PEI - (Push effective indirect address)
($D4 - 2 bytes)

The direct register is added to - forming adrsl. The contents of adrsl,adrsl+1 is then pushed on the stack, high byte first.

PER -- (Push effective relative indirect address)
($62 -- 3 bytes)

The operand gives an offset. This is added to the current program counter to form an address adrsl. The contents of adrsl,adrsl+1 are pushed on the stack, high byte first.

PLB (Pull data bank register from stack)
($AB 1 byte)

The data bank register and program bank register pertain to the extended addressing capabilities of the 65816 and thus this has little use for the 65802.

PHB (Push data bank register onto stack)
($8B 1 byte)

PLD (Pull direct register from stack)
($2B 1 byte)

PHD (Push direct register onto stack)
($0B 1 byte)

PHK (Push program bank register on stack)
($4B 1 byte)

STATUS MANIPULATION INSTRUCTIONS:

REP #- (Reset status bits defined by byte)
($C2 2 bytes)

If bit n of - is 1 then the corresponding n-th status bit is reset. If bit n of - is 0 then the n-th status bit is unchanged.

SEP #- (Set status bits defined by byte)
($E2 2 bytes)

If bit n of - is 1 then the corresponding n-th status bit is set. If bit n of - is 0 then the n-th status bit is unchanged.

XCE (Exchange emulation bit E with carry C)
($FB 1 byte)

Note that this is the only way you can change or read the E emulation flag.

NEW REGISTER MANIPULATION INSTRUCTIONS:

(Recall that the C register is the accumulator A together with B.)

TCD (Transfer C accumulator to direct register D)
($5B 1 byte)

TDC (Transfer direct register D to accumulator C)
($7B 1 byte)

TCS (Transfer accumulator C to stack register)
($1B 1 byte)

TSC (Transfer stack register to accumulator C)
($3B 1 byte)

TXY (Transfer X to Y)
($9B 1 byte)

TYX (Transfer Y to X)
($BB 1 byte)

XBA (Exchange the A and B halves of accumulator)
($EB 1 byte)

NEW BRANCH AND JUMP INSTRUCTIONS:

BRL -- (Branch relative long)
($82 3 bytes)

This is just like a 6502 branch except that the offset can be from -32768 to +32767.

JSR (--,X) (Preindexed jump to subroutine)
($FC 3 bytes)

A jump to subroutine is performed to the address held in the location -- plus X.

(There are also some "long" jumps which do not do anything useful on the 65802, and are not supported by the assembler.)

BLOCK MOVE INSTRUCTIONS:

MVP -,- (Block move forward)
($44 3 bytes)

This moves the byte at the address held in the X register to the address held in the Y register. Then X and Y are incremented and the accumulator C is decremented. The process is repeated until the accumulator is zero. The two bytes following the opcode specify the destination bank and the source bank respectively. On the 65802 these should just be zero, but you must specify them nevertheless.

MVN -,- (Block move backward)
($54 3 bytes)

This moves the byte at the address held in the X register to the address held in the Y register. Then X and Y and the accumulator C are decremented. The process is repeated until the accumulator is zero. The two bytes following the opcode specify the destination bank and the source bank respectively. On the 65802 these should just be zero, but you must specify them nevertheless.

MISCELLANEOUS INSTRUCTIONS:

COP - (Coprocessor)
($02 2 bytes)

Causes a jump to the address held in $FFF4,5. The meaning of the second byte would depend on how this is implemented in hardware. This can be used to call a coprocessor such as an arithmetic processor, but this must be tied to the hardware.

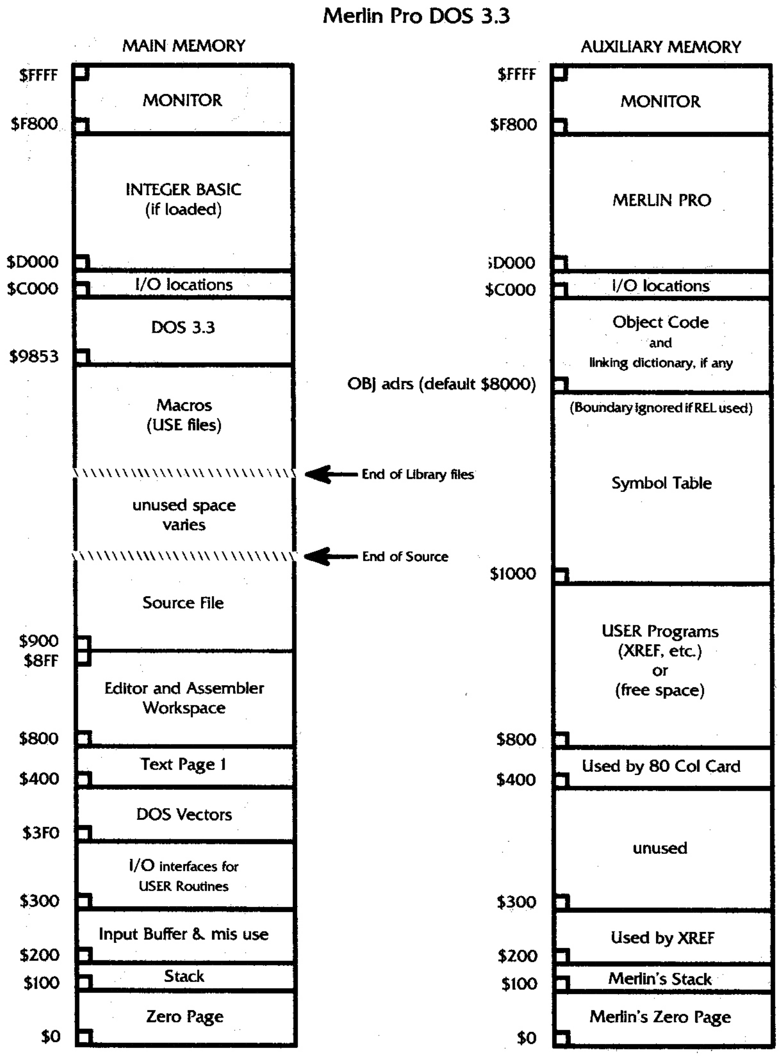
STP (Stop the clock)
($DB 1 byte)

Stops the microprocessor clock.

WAI (Wait for interrupt)
($CB 1 byte)

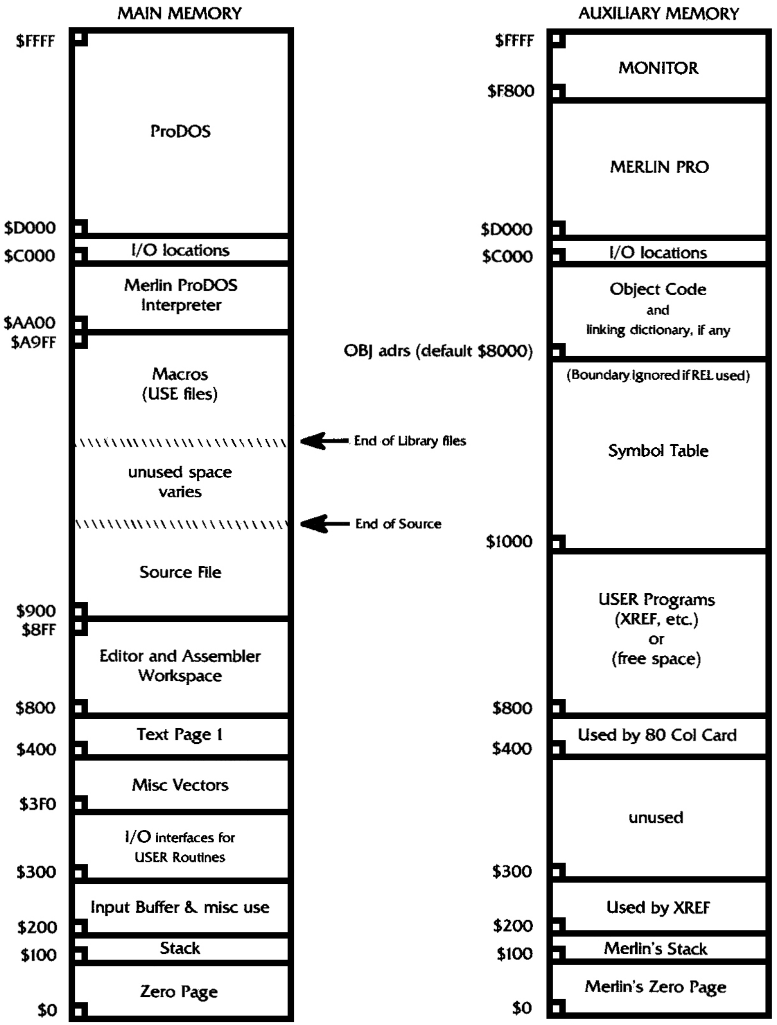
Pulls the RDY line low. This ends when an IRQ or NMI happens.

DOS 3.3 Merlin Pro Memory Map



ProDOS Merlin Pro Memory Map

Merlin Pro ProDOS



[Pro DOS Version]

NEW DISK COMMANDS

There is an alternate way to set the disk prefix. Press D for Disk Command, then enter PFX= or PFX=1 to specify Slot 6, Drive 1, or PFX=2 for Slot 6, Drive 2. You can use the new SLOT command to specify slots other than 6. SLOT is intended to be used with the PFX= and CATALOG command as described below.

CATALOG COMMAND

After using the CATALOG command, if you press =, =1, or =2, Merlin Pro will set the prefix to the volume found in the specified drive and then catalog that volume.

If you press OPEN APPLE during a catalog, Merlin Pro lists only the directory files present in the specified directory.

If you press CLOSED APPLE during a catalog, Merlin Pro lists only the TEXT files present in the specified directory.

If you press OPEN APPLE and CLOSED APPLE simultaneously during a catalog, Merlin Pro lists only the BIN files present in the specified directory. Note that these keys must be pressed and held throughout the entire catalog listing process.

INTERPRETER

If the Merlin Pro ProDOS interpreter cannot find a disk volume required for linking or assembly, it will ask for the correct volume to be inserted. This request can be aborted by pressing CTRL-C or RESET. This only applies to volumes, and not files. Thus, if you want a PUT opcode to prompt you to switch disks, you must use the full pathname with the PUT opcode.

Note that this feature will not work with the Linker when using one disk drive.

If the present prefix does not correspond to any volume online, Merlin Pro will give a VOLUME NOT FOUND error.

The PROGRAM TOO LARGE error message has been changed to MEMORY IN USE.

[DOS 3.3 Version]

The DOS 3.3 version does not perform the same volume checking as the ProDOS version. However, it is possible to simulate this with the following code:

```
          LST
     XXX KBD "INSERT MYFILE DISK AND TYPE 0 <RETURN>"
          PAUSE
```

The assembler will stop at KBD on the first pass and assign a 0 value to XXX (any dummy label you desire). PAUSE will force a pause on the second pass and LST makes sure you will see the KBD line. On the second pass, assembly resumes when you press any key (it is not necessary to type 0 and press RETURN.

[ProDOS and DOS 3.3 Versions]

MERLIN PRO AND "SPEED UP" CARDS

Merlin Pro will work either in main or auxiliary memory (aux is the default). If you are using the main memory version, you will get about a 1.6 speed improvement with the SpeeDemon card, and about a 2x increase with the Accelerator. This is due to the heavy use of auxiliary memory during assembly.

To select the main memory version with DOS 3.3, change the HELLO program to BLOAD MERLIN.X instead of MERLIN.

To select the main memory version with ProDOS, use a $C3 as the fifth byte in the PARMS file. The V-bit of that location is used as a flag to instruct the interpreter to make the main memory modifications.

A + sign after the MERLIN PRO VERSION 2.xx on the EXEC mode screen indicates the main memory version is active.

Some utilities do not work with the ProDOS main memory version. This is because ProDOS is moved to auxiliary memory. Programs that do not switch zero pages will work correctly. Programs designed to be run in 64K will most likely run properly. The Filer and Convert programs will run as long as the -' command is used to run them, and all Merlin Pro utilities will function correctly. The QUIT command moves ProDOS back to main memory.

MACROS

Errors in macros no longer abort assembly.

LINKER

The addresses of all external references are printed whether or not they are resolved.

If you use the TRON command prior to the LINK command, only the errors will be printed in the external list (NOT RESOLVED and DUPLICATE errors).

LUP

In a LUP, if the @ character appears in the label column, it will be increased by the loop count (thus A,B,C ...). Since the loop count is a countdown, these labels will go backwards (the last label has the A). This makes it possible to label items inside a LUP. This will work in a LUP with a maximum length of 26, otherwise you will get a BAD LABEL error and possibly some DUPLICATE LABEL errors.

CLOCK

This utility is an interrupt driven software clock designed for the //c which lacks a clock to do the time stamping available in ProDOS. It requires the //c because it uses the VBLINT interrupt provision. This utility should be used with caution! If it is overwritten, anything can happen and probably will. Press RESET to turn off interrupts. The source files are provided in the SOURCE directory on the ProDOS version.

CONV.LNK.REL [ProDOS only]

This makes Merlin Pro´s REL files compatible with Apple´s RLOAD and RBOOT programs. It will convert a Merlin Pro LNK file to Apple´s REL format (only if there are no externals). You can BRUN it from the EXEC mode. If there is a source file in memory, it will just return, so enter NEW first in the Editor. You will be prompted for the pathname of the file to be converted. The program will do the conversion and set up the converted file for Merlin Pro´s object save command. The CONV.LNK.REL utility does not write anything to disk and does not delete or otherwise damage the original file.

You will be prompted for the pathname of the file you want to convert. The program will do the conversion and set up the converted file for Merlin Pro's object save command. The CONV. LNK.REL utility does not write anything to disk and does not delete or otherwise damage the original file.

CLR.HI.BIT [ProDOS only]

This converts a source file in memory to positive Ascii so the file can be sent to other programs that expect data in this form, such as Apple's ProDOS ED/ASM. To use it, just BRUN UTIL/CLR.HI.BIT and then save the source. CAUTION: If you reenter the Editor, the source will be deleted from memory, since the Editor does not like this format.

65C02 SPECIAL NOTES

To assemble or disassemble 65C02 code with the older //e ROMs, you must first BRUN MON.65C02. This must be done from BASIC if you are using the DOS 3.3 version. This utility is not needed with the newer //e or //c ROMs.

Whether you are using the ProDOS or DOS 3.3 version, you MUST use the XC opcode as the very first line in your code. This serves as a flag to tell Merlin Pro that you are using 65C02 opcodes.

MANUAL CORRECTIONS and ADDITIONAL INFORMATION

(Page 109)

Configuration (ProDOS version)

Configuration data is kept in a file called PARMS which is loaded when the assembler is run. To change the data in the source file called PARMS.S, with the prefix set to /MERLIN/, type L to Load Source. Then type SOURCE/PARMS at the prompt. When you are done making changes, reassemble the file. Use S to SAVE the source code as /MERLIN/SOURCE/PARMS (Merlin Pro adds the .S automatically). Then save the object code as /MERLIN/PARMS by using the O command.

MERLIN PRO™

The Professional Macro Assembler for the Apple IIe and IIc

MERLIN PRO is an extremely powerful and comprehensive macro assembler designed specifically for the 128K Apple IIe or IIc. With all of the regular Merlin features, this professional version also offers additional enhancements for the serious programmer working with either DOS 3.3 or ProDOS.

The MERLIN PRO system consists of four integrated co-resident modules plus many auxiliary and utility programs. The four main modules are:

- EXECUTIVE MODE which provides file management and disk I/O operations such as Save Object Code, Load or Save Source Code, Read or Write Text File, Append File, Change Drive, and also includes a special ProDOS interpreter.
- EDITOR MODE for writing or editing programs with over 40 word processor commands such as Add, Edit, Insert, Delete, Copy, Move, Global Search and Replace and more! Also includes commands for formatted printouts with headers and page boundary breaks.
- ASSEMBLER MODE with sophisticated features such as macros, macro libraries, nested macros, conditional assembly, assemble to disk, linked files, dummy program segments and more.
- RELOCATING LINKER to automatically generate relocatable object code, library routines, run time packages and so on.

MERLIN PRO offers over 50 Psuedo Opcodes for true programming flexibility. It also allows the use of Local Labels and supports both Entry and External Label Definitions for use with the Relocating Linker. MERLIN PRO not only assembles 6502 programs but also supports 65C02 and 65802 opcodes.

MERLIN PRO also includes many utilities and support programs such as:

- SOURCEROR
  A sophisticated and easy to use disassembler that creates MERLIN PRO source files out of binary programs. Sourceror uses a predefined Applesoft Source label file to give the most detailed listings possible. The label file can also be edited to include your own labels.
- APPLESOFT SOURCE
  This utility creates a fully labeled and commented listing of Applesoft BASIC. This is an invaluable reference for anyone attempting to gain a better understanding of the internal workings of Applesoft. Provides source listings for all versions of Applesoft including Apple II, II+, IIe or IIc.
- MACRO LIBRARIES
  Libraries of commonly used macro definitions and fundamental operations such as floating point routines, RWTS routines, Rockwell 65C02 bit operations, and more.

MERLIN PRO is compatible with the Apple IIe and IIc in both 40 and 80 column formats, supports upper/lower case entry, and is hard disk compatible.

SYSTEM REQUIREMENTS:
128K Apple IIe or IIc

Roger Wagner™
PUBLISHING, INC.

ISBN 0-927796-04-X