MERLIN™

The Macro Assembler For The Apple II Family

By Glen Bredon

Roger Wagner™
PUBLISHING, INC.

MERLIN™

The Macro Assembler For The Apple
By Glen Bredon

What kind of assembler would you expect from a company that established itself with programming utilities? Simply the best, of course. **MERLIN** is more than just an assembler. It is an extremely powerful macro assembler, with a sophisticated editor, combined with numerous other files and programming utilities into a truly remarkable package.

The **MERLIN** assembler, besides having the common features you would expect, allows such enhancements as the optional writing of object files directly to disk, and linking files to assemble source listings otherwise too large to fit in memory at once. The source listing can also use macro routines, thus saving time and space. The macro feature of **MERLIN** allows you to give a simple name to often used routines in a listing, and then enter only the name of the macro when entering text.

MERLIN will read and write text files as well as binary source files, and is often capable of using files generated on other assemblers with little or no adjustments. The global search/replace function of the editor also makes it easy to change pseudo-ops that may have been peculiar to the other assembler.

MERLIN supports BOTH 6502 and 65C02 opcodes. Additionally, **MERLIN** supports **SWEET 16** opcodes as well, and the manual includes a short tutorial on this subject by Steve Wozniak, co-founder of Apple Computer, Inc.

In addition to the **MERLIN** assembler, the package also includes:

SOURCEROR:

This generates pseudo source code from raw binary data. Uses a pre-defined Applesoft Source label file to give the most detailed listings possible. The label file can also be edited to include your own labels as you desire.

MACRO LIBRARY:

A library of commonly used macro definitions and fundamental operations such as multiplication and divide routines is included on the diskette.

SWEET 16 SOURCE:

A source code for a transportable SWEET 16 interpreter, usable even without the Integer non-Auto Boot ROM.

APPLESOFT SOURCE:

If you have Applesoft in ROM or LANGUAGE CARD, you can use utilities included in the **MERLIN** package to create a fully labeled and commented listing of Applesoft BASIC. This is an invaluable aid to anyone attempting to gain a deeper understanding of the internal workings of Applesoft.

MERLIN is compatible with most 80 column cards and supports upper/lower case entry, including the one-wire shift key mod and commercial lower case devices.

MERLIN is shipped on a double-sided DOS 3.3 diskette and is hard disk compatible.

SYSTEM REQUIREMENTS:
Apple II/II+ with language/RAM card or Apple IIe or Apple IIc

Roger Wagner™
PUBLISHING, INC.

ISBN 0-927796-03-1

MERLIN™

The Macro Assembler For The Apple

By Glen Bredon

INSTRUCTION MANUAL

ISBN 0-927796-03-1

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.

First, our legal stuff...

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 you 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 your personal use in connection with a single computer. You may not distribute copies of, or any part of, this product without the express written permission of Roger Wagner Publishing, Inc.

LIMITATION ON WARRANTIES AND LIABILITY

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

Then Apple's...

DOS 3.3 Standard is a copyrighted program of Apple Computer, Inc. licensed to Roger Wagner Publishing, Inc. to distribute for use only in combination with Merlin.

APPLE COMPUTER, INC. MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, REGARDING THE ENCLOSED SOFTWARE PACKAGE, 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, prefering the solitude of the Sierra Nevada mountains where he has helped establish wilderness reserves.

PRODUCT REFERENCE MERLIN 2M1084LC

TABLE OF CONTENTS

INTRODUCTION

Assembly Language Whys and Wherefores

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

Computer languages are often referred to as "high level" or "low level" languages. BASIC, COBOL, FORTRAN and PASCAL are all high level languages. A high level language is one that 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 available memory.

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

The 65Ø2 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 65Ø2 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 65Ø2 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 standard Apple II has a built-in mini-assembler that permits simple Assembly Language programming.

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

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

Before using this or any other assembler, the user is expected to be somewhat familiar with the 65Ø2 architecture, modes of addressing, &c. This manual is not intended to teach Assembly Language programming. Many good books on 65Ø2 Assembly programming are available at your local dealer; some are referenced later in this section.

Backgrounds and Features

MERLIN is a "Ted-based" editor-assembler. This means that while it is essentially new from the ground up, it adheres to and follows almost all of the conventions associated with TED II+, in terms of the command mnemonics, pseudo-ops, &c.

The original TED ASM was written by Randy Wiggington and Gary Shannon. It has been widely distributed "under the counter" by user groups and individuals, under many names, and in a variety of versions. Seemingly, each person added his own enhancements and improvements. MERLIN is no exception. Representing a major step forward, with the addition of macro capability, MERLIN appears on the scene now as one of the most advanced and sophisticated editor-assemblers for the Apple II, yet retains all of the easy-to-use features of TED that make it desirable to a beginner in assembly language programming.

Significant changes incorporated in MERLIN, in addition to macros, include the use of the logical operators AND, OR, and EOR, the math operator for division, the ability to list with or without line numbers, and substantially faster editing. Similarly, the edit module now includes many additional commands to facilitate editing, and the "Read" command allows any Apple text file to be read into the edit buffer, thus permitting the use of source files from other assemblers, such as the DOS Tool Kit's.

MERLIN assumes that your system has at least 48K memory and operates under DOS 3.3. BEWARE of "custom" DOS's. MERLIN does an automatic MAXFILES 2 upon entry, then reverts to the usual value on exit.

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 65ØØ-2Ø.

PROGRAMMING THE 65Ø2 - Rodnay Zaks, Sybex C-2Ø2.

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

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

FLOATING POINT PACKAGE - Apple Computer, Inc., The Wozpak II

FLOATING POINT LINKAGE ROUTINES - Don Williams, Peeking at Call-Apple Vol I

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 65Ø2 command, but also a brief discussion of its most common uses along with concise, illustrative listings.

CONVERTING BRAND X TO BRAND Y - by Randall Hyde
Apple Orchard, Volume 1, No.1, March/April 8Ø. Useful notes and cross references on converting among assemblers.

CONVERTING INTEGER BASIC PROGRAMS TO ASSEMBLY LANGUAGE
by Randall Hyde
Apple Orchard, as above.

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

MACHINE TOOLS
Call-APPLE in Depth, No. 1

SYSTEM REQUIREMENTS

* 48K APPLE][(16K RAM CARD for 64k MERLIN)
  or APPLE //e
* 8Ø COLUMN BOARD (optional)
* LOWER CASE BOARD (optional)

Hardware Compatibility List

* ALS SMARTERM 8Ø COLUMN BOARD
* APPLE //e 8Ø COLUMN BOARD
* FULL-VIEW 8Ø - 8Ø COLUMN BOARD
* M & R SUP´R´TERMINAL 8Ø COLUMN BOARD
* MICROMAX VIEWMAX 8Ø - 8Ø COLUMN BOARD
* VIDEX ULTRATERM
* VIDEX VIDEOTERM
* VISTA VISION 8Ø - 8Ø COLUMN BOARD
* WIZARD 8Ø - 8Ø COLUMN BOARD

* ANDROMEDA 16K BOARD
* MICROSOFT 16K RAM BOARD
* OMEGA MICROWARE RAMEX 16 - 16K RAM BOARD

NOTE: MERLIN has been tested with the cards/boards listed above. The author makes no guarantees with respect to the operation of MERLIN with any 8Ø column boards not listed.

BEGINNERS GUIDE TO USING MERLIN

By T. Petersen

Notes and demonstrations for the beginning MERLIN programmer.

Introduction

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

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

A note of clarification:

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

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

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

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

EDITOR CONTROL MODE
Editor Module
Assembler Module
Symbol Table Generator Module

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

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

Input

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

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

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

Steps from the very beginning:

1. BRUN MERLIN or 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 'OBJ', space again, type '$3ØØ', and hit RETURN. Note that in most cases the 'OBJ' pseudo-op is neither required nor desirable.

7. On line 3, perform the same sequence but for ORG: space, type 'ORG', space, type '$3ØØ', RETURN.

The above two steps instruct the assembler to place the following program both physically (with OBJ) and logically (with ORG) at $3ØØ.

8. On line 4, 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 '$FBDD' (making 'BELL' in effect a mnemonic). Also, if the location of BELL were to change, all that needs changing is the 'EQU' statement, and not a mess of '$FBDD's.

9. Line 5 - 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.

1Ø. Line 6 - ´DONE´, space, ´RTS´, RETURN.

11. 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.

12. The screen should now appear like this:

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

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

Field Number...

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

Two is reserved for opcodes, such as the MERLIN pseudo-opcodes OBJ, ORG and EQU, and the 65Ø2 opcodes JSR and RTS.

Three is for operands, such as $3ØØ, $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 before a label. Press space once after label (or if there is no label, once after the line number) for the opcode.
2) Space once after the opcode for the operand. Space once after the operand for the comment. If there is no operand, type a space and a semicolon.

System and Entry Commands

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

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

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' (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.

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          OBJ   $3ØØ
     3          ORG   $3ØØ
     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 subsequent lines renumbered. The listing appears the same as in the subsection on INPUT, Step 12.

This automatic renumbering feature makes it IMPERATIVE that when 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 other sub-commands (CTRL-characters) used under the EDIT command function similarly. Read the definitions in Section 3 and practice a few operations.

Assembly

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

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

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

Type N, and you will see:

```
ASSEMBLING
                      1    *DEMO PROGRAM 1
                      2          OBJ  $3ØØ
                      3          ORG  $3ØØ
                      4    BELL  EQU  $FBDD
Ø3ØØ     2Ø DD FB     5    START JSR  BELL  ;RING THE BELL
Ø3Ø3     6Ø           6    DONE  RTS

--END ASSEMBLY, 4 BYTES, ERRORS: 0
```

```
SYMBOL TABLE - ALPHABETICAL ORDER

     BELL      =$FBDD          ?    DONE          =$Ø3Ø3

?    START     =$Ø3ØØ

SYMBOL TABLE - NUMERICAL ORDER

?    START     =$Ø3ØØ          ?    DONE          =$Ø3Ø3

     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 numbers down the middle of the screen is the now familiar, formatted source code.

To the left of the 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 OBJ and ORG in the section entitled "Pseudo Opcodes - Directives"). After the colon is the number '2Ø'. 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 OBJ and 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 65Ø2 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-labelling convention which indicates the subject file is source code. This suffix is automatically appended to the name by the SAVE SOURCE command.

3. Hit RETURN to return to EXEC mode and input ´O´, for OBJECT CODE SAVE. The object file should be saved under the same name as was earlier specified for the source file, so press "Y" to accept ´DEMO1´ as the object name. There is no danger of overwriting the source file because 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´), type ´MON´, RETURN and the monitor prompt ´*´ appears. Enter ´3ØØG´, 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 to any DOS 3.3 diskette using the FID utility program from Apple´s System Master Diskette.

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

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.

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 commands 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 at this time 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

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.

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 $8ØØØ by default) and not that of the actual current location of the assembled code (which is $8ØØØ or whatever OBJ you have used). 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 ($3ØØ,$8ØØ, &c). There is usually no need to use an OBJ in the source code, unless the object code will be too long for the space available at $8ØØØ and above.

Q:QUIT

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, if it is more convenient than the one provided by "C".

R:READ TEXT FILE

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

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

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 $2Ø to $4Ø. 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

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.

THE EDITOR

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

Command Mode

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

When the syntax for each command is given, parentheses "()" indicate a required value. When the value or character is optional, angle brackets "<>" are used.

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

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

Control-L toggles the current case of alphabetic input anywhere within the editor. If you are in upper case, typing CTRL-L will place you in lower, and vice versa. Upper case is defaulted to when entering each new line. If your Apple is not capable of displaying lower case, then all lower case text will be displayed as upper case flashing. To change the case of a word, type CTRL-L, then copy over the word using the right arrow.

Hex-Dec Conversion

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, which is mainly convenient for the HIMEM: and SYM commands.

HImem:

HI: (address)

HImem: sets the upper limit for the source file and beginning address for the OBJ file (default OBJ address). HIMEM defaults to $8ØØØ, and does not have to be set unless you use a non-default object address.

NEW

Deletes the present source file in memory, resets HIMEM to $8ØØØ and starts fresh.

PR#

PR#(Ø-7)

Same function as in BASIC. Mainly used for sending an editor or assembly listing to a printer. DO NOT use this to select an 8Ø-column card. NOTE: that PR# is automatically turned off after an ASM command, but not after a LIST or PRINT command.

USER

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 user defined printer drivers. (You must be careful that your printer driver does not use zero page addresses, except the I/O pointers and $6Ø - $6F, because this is likely to interfere with MERLIN's heavy zero page usage).

TABS

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

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

LENgth

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

Where

Where (line number)

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

MONitor

This exits to the monitor. You may re-enter 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. This facility is designed for experienced Apple programmers, and is not recommended to beginners.

You may also re-enter the editor directly with a ØG. This re-entry, unlike the others, will use the zero page pointers at $ØA - $ØF 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.

TRuncON

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 4Ø column screen. In the assembler, when used as a pseudo-op, it limits printing of the object code to three bytes per line and has no effect on comments.

TRuncOFf

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

Exits to EXEC mode.

ASM

This passes control to the assembler, which attempts to assemble the source file. First, however, you are asked if you wish to "update the source". This is to remind you to change the date or identification number in your source file. If you answer "N" then the assembly will proceed. If you answer "Y", you will be presented with the first line in the source containing a "/" and are placed in EDIT mode. When you finish editing this line and hit RETURN, assembly will begin. If you use the 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 "/".

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.

Delete

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

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

Replace

Replace (line number)
Replace (range)

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

List

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

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

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. Any other key will restart it. This space bar pause also works during assembly and the symbol table print out.

. [period]

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

/

/ <line number>

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

Print

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

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

PRinTeR

PRinTeR (command)

This command is for sending a listing to a printer with page headers and provision for page boundary skips. (The default parameters may be set up using the configure program included on the MERLIN diskette.) The syntax of this 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 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, or CTRL-I8ØN for most Apple cards.

PRTR Ø (no strings required here) will allow you to see where the page breaks occur. If an 8Ø 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>
```

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

Change

```
Change (d-string d-string)
Change (line numbers) <d-string d-string>
Change (range) <d-string d-string>
Change (range list) <d-string d-string>
```

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

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

COPY

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

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

MOVE

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

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

Edit

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

This presents each line of the line number, range, 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

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

FIX

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

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

SYM

SYM (address)

MERLIN normally places the symbol table on the language card (in bank 1 of $DØØØ-$DFFF). This space is quite adequate for all but gigantic programs. In case this space is used up, the SYM command gives you a means to direct the assembler to continue the symbol table in another area. If you type SYM $9ØØØ, for example, and assemble the program, when and if the symbol table uses up its normal space, it will be continued at $9ØØØ until it reaches BASIC HIMEM.

The SYM address must be equal to or above MERLIN's HIMEM and below BASIC HIMEM. If the symbol table grows beyond the allotted space, you will get a MEMORY FULL error during the first pass of assembly.

NOTE: The SYM address you specify will be cancelled by a (MERLIN) HIMEM: command or by an exit to EXEC mode and re-entry (set HIMEM: before setting up a SYM address).

VIDeo

VIDeo (slot)

This command is designed to select or deselect an 8Ø 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 8Ø column board! PR# is designed for selection of a printer ONLY. An 8Ø column board in slot 3 for example, can be selected by typing, from the editor: VIDEO 3.

It is deselected by VIDEO Ø or VIDEO $1Ø (or 16) possibly followed by RESET. These two forms both select the standard Apple screen, but VIDEO Ø will cause all lower case output to the screen to be converted to upper case except lower case in the source file will be converted to flashing upper case (output to a printer is never converted). If you have a lower case adapter, you will want to use VIDEO $1Ø (or VIDEO 16) instead of VIDEO Ø when selectng the Apple screen.

If your 8Ø column card has a software screen switch via an escape sequence, this may be used to return to 4Ø column mode. This will be equivalent to "VID $1Ø" and would have to be followed by a VID Ø if you don't have a lower case adapter. For example, use ESC CTRL-Q RETURN on the Smarterm or ESC-Q-CTRL-X on the Sup"R"term.

FW (Find Word)

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

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

Therefore, FW"CAT" will find:

CAT
CAT-1
(CAT,X)

but will not find CATALOG or SCAT.

CW (Change word)

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>

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

EW (Edit word)

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

This is to EDIT as FW is to FIND.

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

VAL

VAL "expression" [like, Oh my God! Fer sure!]

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.

Examples of the use of VAL:

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

Add/Insert Modes

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

Add

The Add command places you in the ADD mode, and acts much like entering additional BASIC lines with auto line numbering. However, you may enter lower case text (useful for comments if you have a lower case adapter) by typing CTRL-L. This acts as a case toggle, so another CTRL-L returns you to UPPERCASE mode. 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)

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

Edit Mode

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

The EDIT commands and functions are very similar, but not identical to those in Neil Konzen's excellent Program Line Editor and Southwestern Data System'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.

Control-D (delete)

Deletes the character under the cursor. It can also be referred to as a backwards delete. The DELETE key of the Apple //e also accomplishes the same action.

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).

Besides enabling the insertion of control characters, CTRL-O also allows the user to type characters not normally available on the Apple keyboard.

Control-O followed by:

<	gives	Control _
>	"	Control \
K	"	[
L	"	\
M	"	]
N	"	^
O	"	
k	"	{
l	"	|
m	"	}
n	"	~
o	"	(whatever $FF gives on your machine)

NOTE: If you are using a shift key modification, depending on which one you have, shift-M may give uppercase M and you will have to use CTRL-O to get the right bracket.

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 3Ø 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 (delete line right)

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

Return (RETURN key)

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

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

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

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

Number Format

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

```
LDA #1ØØ     LDA #$64     LDA #%11ØØ1ØØ     LDA #%Ø11ØØ1ØØ
```

As indicated, leading zeros are ignored. The "#" here stands for "number" or "data", and the effect of these instructions is to load the accumulator with the number (decimal) 1ØØ.

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

```
LDA 1ØØØ     LDA $3E8     LDA %111111Ø1ØØØ
```

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

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

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

is a good deal more mysterious that its decimal equivalent:

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

Source Code Format

A line of source code typically looks like:

```
    LABEL OPCODE OPERAND ;COMMENT
```

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

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

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

The assembler examines only the first 3 characters of the OPCODE (with certain exceptions such as the Sweet 16 opcode 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 truncate the opcode to seven letters and will not look well with one longer than four unless there is no operand.

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

Expressions

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

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

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

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

Some examples of legal expressions are:

```
LABEL1-LABEL2        (LABEL1 minus LABEL2)
2*LABEL+$231         (2 times LABEL plus hex 231)
1234+%1Ø111          (1234 plus binary 1Ø111)
"K"-"A"+1            (ASCII "K" minus ASCII "A" plus 1)
"Ø"!LABEL            (ASCII "Ø" EOR LABEL)
LABEL&$7F            (LABEL AND hex 7F)
*-2                  (present address minus 2)
LABEL.%1ØØØØØØØ      (LABEL OR binary 1ØØØØØØØ)
```

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

Immediate Data

For those opcodes such as LDA, CMP, &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 65Ø2 does not accept 2-byte DATA)
#/expression	is optional syntax for the high byte of the expression

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

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

Addressing Modes (65Ø2 Opcodes)

The assembler accepts all the 65Ø2 opcodes with standard mnemonics. It also accepts BLT (Branch if Less Than) and BGE (Branch if Greater or Equal) as pseudonyms for BCC and BCS, respectively.

There are 12 addressing modes on the 65Ø2. The appropriate MERLIN syntax for these are:

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

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

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

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

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

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

Sweet 16 Opcodes

The assembler accepts all Sweet 16 opcodes with the standard mnemonics. The usual Sweet 16 registers RØ to R15 do not have to be "equated" and the "R" is optional. TED II+ users will be glad to know that the SET opcode works as it should, with numbers or labels. 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 R3 LABEL. It should be noted that the NUL opcode is assembled as a one-byte opcode (the same as HEX ØD) and not a two byte skip as this would be interpreted by ROM Sweet 16. This is intentional, and is done for internal reasons.

Pseudo Opcodes - Directives

EQU (=) (EQUate)

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

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

ORG (set ORiGin)

ORG expression

Establishes the address at which the program is designed to run. It defaults to the present value of HIMEM: ($8ØØØ by default). Ordinarily there will be only one ORG and it will be at the start of the program. If more than one ORG is used, the first one establishes the 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.

OBJ (set OBJect)

OBJ expression

Establishes the address at which the object code will be placed during assembly. It defaults to MERLIN's HIMEM. There is rarely any need to use this pseudo-op and inexperienced programmers are urged not to. An OBJ above BASIC HIMEM (or the SYM address, if any) will defeat generation of object code. This may be used when sending a long listing to a printer or when using direct assembly to disk (opcode DSK).

PUT (PUT a text file in assembly)

PUT filename

"PUT filename" (drive and slot parameters accepted in standard DOS syntax) reads the named file with the "T." prefix included unless the filename starts with a character less than "@" and "inserts" it at the location of the opcode.

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

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

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

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

VAR (setup VARiables)

```
 VAR expr;expr;expr...
```

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

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

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

The SAVe command sets the address of the saved file to its correct value. For example, if your program contains three SAV commands, then it will be saved in three pieces. When BLOADed later, they will go to the correct locations, the third following the second and that following the first.

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

DSK (assemble directly to DiSK)

DSK filename

"DSK filename" will direct the assembler to assemble the following code directly to disk. If DSK is already in effect, the old file will be closed and the new one begun. This is useful primarily for extremely large files. For moderately sized programs, SAV is preferred since it is 30% faster and theoretically more reliable.

END (END of source file)

```
 END
```

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

DUM (DUMmy section)

```
 DUM expression
```

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

DEND (Dummy END)

```
 DEND
```

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

Sample usage of DUM and DEND:

```
 1          ORG  $1000
 2
 3 IOBADRS =     $B7EB
 4
 5          DUM  IOBADRS
 6 IOBTYPE  DFB  1
 7 IOBSLOT  DFB  $60
 8 IOBDRV   DFB  1
 9 IOBVOL   DFB  0
10 IOBTRCK  DFB  0
11 IOBSECT  DFB  0
12          DS   2          ;pointer to DCT
13 IOBBUF   DA   0
14          DA   0
15 IOBCMD   DFB  1
16 IOBERR   DFB  0
17 ACTVOL   DFB  0
18 PREVSL   DFB  0
19 PREVDR   DFB  0
20          DEND
21
22 START    LDA  #SLOT
23          STA  IOBSLOT
24 *  And so on
```

Formatting

LST ON/OFF (LiSTing control)

LST ON or OFF

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 ERINE will have the same effect as LST OFF. The LST directive will have no effect on the actual generation of object code. If the LST condition is OFF, the object code will be generated much faster, but this is recommended only for debugged programs.

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

EXP ON/OFF (macro EXPand control)

EXP ON or OFF

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

PAU (PAUse)

PAU

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

PAG (new PAGe)

PAG

This sends a formfeed ($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

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 AST 1Ø will send (decimal) 1Ø asterisks, for example. The number is treated modulo 256 with Ø being 256 asterisks! This differs from TED II+, which recognizes the operand as a hex expression, so any AST statements in a TED II+ source will need to be converted.

SKP (SKiP lines)

SKP expression

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

TR ON/OFF (TRuncate control)

TR ON or OFF

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

Strings

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

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

ASC (define ASCii text)

ASC dstring

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. 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.

DCI (Dextral Character Inverted)

DCI d-string

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

INV (define INVerse text)

INV d-string

This puts a delimited string in memory in inverse format (more specifically, with the 7th bit clear). All choices of delimiter have the same effect on the 8th bit as with ASC.

FLS (define FLaShing text)

FLS d-string

This puts a delimited string in memory in flashing format (that is, with the 7th bit set). All choices of delimiter have the same effect on the 8th bit as with ASC.

REV (REVerse)

REV d-string

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.

Data and Allocation

DA (Define Address)

DA expression

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

DA $FDFØ gives (hex) FØ FD

DA also accepts multiple data separated by commas (such as DA 1,1Ø,1ØØ).

DDB (Define Double-Byte)

DDB expression

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

DFB (DeFine Byte)

DFB expression

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.

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 $34,1ØØ,LAB1-LAB2,%1Ø11,>LAB1-LAB2
```

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

```
34 64 DE ØB Ø9
```

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

HEX (define HEX data)

HEX hex-data

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

Examples of HEX:

```
HEX ØØØØFFFF
HEX ØØ,ØØ,FF,FF
HEX 281A8F544EFFED
```

DS (Define Storage)

DS expression

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

KBD (define label from KeyBoarD)

label KBD

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

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.

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

```
        ]A    =   Ø
              LUP $FF
        ]A    =   ]A+1    (Note: will not work inside a Macro)
              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.

CHK (place CHecKsum in object code)

```
 CHK
```

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 is used to verify the existence of an accurate image of the program in memory.

ERR (force ERRor)

```
 ERR expression
```

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

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

```
     ERR  *-1/$96ØØ
```

NOTE: The above example would only alert you that the program is too long, and will not prevent writing above $96ØØ during assembly, but there can be no harm in this, 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  ($3ØØ)-$4C
```

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

USR (USeR definable op-code)

USR optional expressions

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    = $28Ø   ;contains the operand and
                   ;comment in positive ASCII
```

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

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

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

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

Appropriate locations for your routine are $3ØØ-$3CF and $8AØ-$8FF. You must not write to $9ØØ. For a longer routine, you may use high memory, just below $9853. If you are sure that the symbol table will not exceed $1ØØØ bytes, you could use the SYM EDITOR command to protect your routine from overwrite by the object code. SYM would have to be set at least one byte below your code.

You may use zero page locations $6Ø-$6F, but should not alter other locations. Also, you must not change anything 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).

To gain further understanding of the use of USR, read the source file SCRAMBLE.S or, for a more sophisticated example, the file FLOAT.S. The first of these uses the USR opcode to put an ASCII string into the object code in a scrambled format. The second is a somewhat complicated routine that uses Applesoft to compute the packed (five-byte) form of a specified floating point number, and put it in the object code. Here, the latter can be used for assembly only on an Apple][Plus.

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 $31Ø then:

```
ERR ($31Ø)-$6Ø
```

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.

Conditionals

DO (DO if true)

DO expression

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

It is also useful for sources designed to generate slightly different code for different situations. For example, if you are designing a program to go 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 Ø), then assembly will not resume until its corresponding FIN is encountered, or an ELSE at this level occurs. Nested DO/FIN structures are valuable for putting conditionals in MACROS.

ELSE (ELSE do this)

```
ELSE
```

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

IF (IF so then do)

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

This checks to see if char is the leading character of the replacement string 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
```

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

Example of the use of conditional assembly:

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

MOVD      MAC
          MOV  ]1;]2
          IF   (,]1              ;Syntax MOVD (ADR1),Y;????
          INY
          IF   (,]2              ;  MOVD (ADR1),Y;(ADR2),Y
          MOV  ]1;]2
          ELSE                   ;  MOVD (ADR1),Y;ADR2
          MOV  ]1;]2+1
          FIN
          ELSE
          IF   (,]2              ;Syntax MOVD ????;(ADR2),Y
          INY
          IF   #,]1              ;  MOVD #ADR1;(ADR2),Y
          MOV  ]1/$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
```

Macros

MAC (begin MACro definition)

Label MAC

This signals the start of a MACRO definition. It must be labeled with the macro name. The name you use is then reserved and cannot be referenced by things other than the PMC pseudo-op (things like DA NAME will not be accepted if NAME is the label on MAC). However, the same thing can be simulated by preceding the MACRO with LABEL EQU *, or LABEL DS Ø, &c. See the section on MACROS for details of the usage of macros.

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 (>>>)

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

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

MACROS

Defining a Macro

A macro definition begins with the line:

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

with Name in the label field. Its definition is terminated by the pseudo-op EOM or <<<. The label you use as Name cannot be referenced by anything other than PMC NAME (or >>> NAME).

You can define the macro the first time you wish to use it in the program. However, it is preferable (and required if the macro uses variables) to first define all macros at the start of the program with the assembly condition off (DO Ø) and then refer to them when needed.

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 PMC (or >>>).

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

Labels inside macros are updated each time PMC (or >>>) 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 invocation (with PMC or >>>) rather than the line inside the macro where the error occured.

Nested Macros

Macros may be nested to a depth of 15. For nesting, macros MUST be defined with the DO condition off. That is, nested macros may NOT be defined the first time they are used (as described above).

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.)

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

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

      DO  Ø
POKE  MAC
      LDA #]2
      STA ]1
      <<<
HTAB  MAC
      >>> POKE.CH;]1
      <<<
      FIN
```

MACRO names may also be put in the opcode column, without using the PMC or >>>, with the following restriction:

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

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

MERLIN with the 65C02 modifications installed does NOT accept this alternate syntax for invocation.

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:

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

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

If, as above, some of the special variables are used in the MACRO definition, then values for them must be specified in the 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 OPERAND 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:

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

It will also accept:

```
        DO   Ø
 PRINT  MAC
        JSR  SENDMSG
        ASC  ]1
        BRK
        <<<
        FIN
        >>>  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.)

A previous version of MERLIN did not have this capability and used commas rather than semicolons in PMC statements. If you had that previous version, a program called "CONVERT" has been provided which changes these commas to semicolons in a matter of seconds. With the source file in memory, it should be BRUN from the EXEC mode´s "COMMAND:" after Catalog.

Sample Program

Here is a sample program intended to illustrate the usage of macros with non-standard variables. It would, however, be simpler and more pleasing if it used]1 instead of]MSG (in which case the variable equates should be eliminated and the values for]1 specified in the >>> lines.)

```
HOME    EQU     $FC58
COUT    EQU     $FDED
DOS     EQU     $3DØ

        DO      Ø           ;Assembly off
SENDMSG MAC                 ;Start of definition of the
                            ;macro "SENDMSG"

        LDY     #Ø
LOOP    LDA     ]MSG,Y      ;Get a character
        BEQ     OUT         ;End of message
        JSR     COUT        ;Send it
        INY
        BNE     LOOP        ;Back for more

OUT     <<<                 ;End of macro definition and
                            ;exit from routine
        FIN                 ;Turn assembly ON
        JSR     HOME        ;Clear screen
]MSG    EQU     LOOP        ;Print msgs...
        >>>     SENDMSG
]MSG    EQU     IMSG
        >>>     SENDMSG
]MSG    EQU     NMSG
        >>>     SENDMSG
        JMP     DOS         ;All done, exit gracefully

FMSG    FLS     "THIS IS A FLASHING MESSAGE"
        HEX     8D8DØØ
IMSG    INV     "THIS IS A MESSAGE IN INVERSE"
        HEX     8D8DØØ
NMSG    ASC     "THIS IS A NORMAL MESSAGE"
        HEX     8D8DØØ
```

The Macro Library

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

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

TECHNICAL INFORMATION

The source is placed at START OF SOURCE when loaded, regardless of its original address.

The important pointers are:

START OF SOURCE	in	$A,$B	(set to $9Ø1 unless changed by CHRGEN7Ø or other)
HIMEM:	in	$C,$D	(defaults to $8ØØØ)
END OF SOURCE	in	$E,$F	

When you exit to BASIC or to the monitor, these pointers are saved on the RAM card at $EØØA-$EØØF. They are restored upon 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#Ø and IN#Ø 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.

General Information

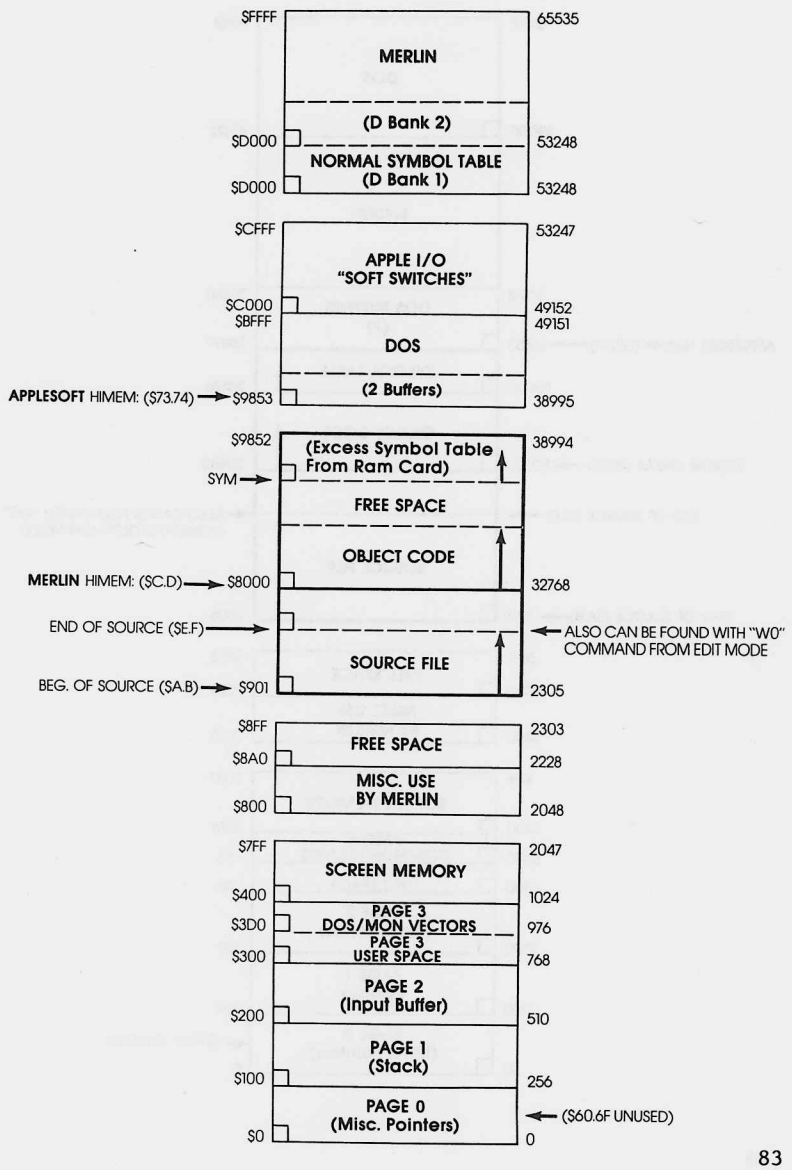
Re-entry after exit to BASIC is made by the "ASSEM" command. Simply use "ASSEM" wherever a DOS command is valid (for example, at the BASIC prompt). A BRUN MERLIN or a disk boot will also provide a warm re-entry and will not reload MERLIN if it is already there. A reload may be forced by typing BRUN BOOT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

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

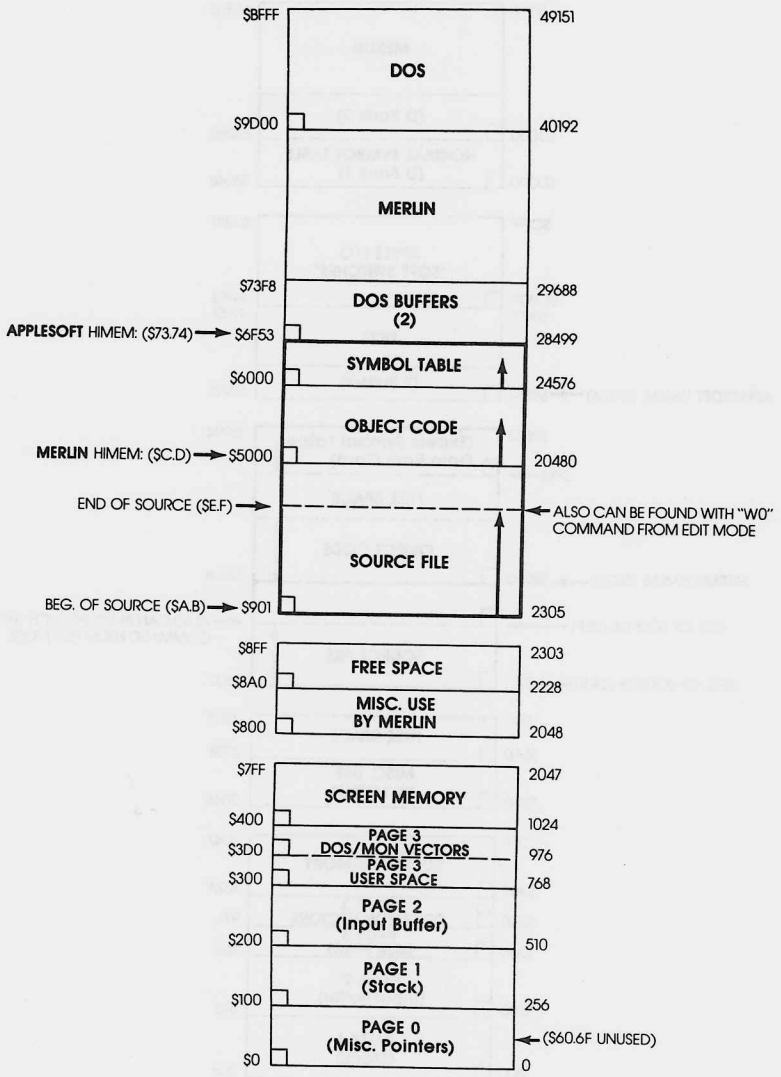
The default ORG and OBJ addresses equal the present value of MERLIN's HIMEM:. It is illegal to specify an OBJ address that is less than HIMEM except that a page 3 address is allowed. If a page 3 ($3ØØ-$3FF) OBJ address is used, the user MUST be careful that the file will not write over the DOS jumps at $3DØ-$3FF as the assembler does NOT check for this error.

If during assembly the object code exceeds BASIC HIMEM (or the SYM address, if one has been specified) then the code will not be written to memory, but assembly will appear to proceed 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. Therefore, if a listing for a very long file is desired, without actually creating code, the user can assemble over DOS and up (OBJ $AØØØ will do).

MERLIN MEMORY MAP
(RAM CARD VERSION)



MERLIN MEMORY MAP
(48K Version)



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

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

MD	=	bit 5
M	=	bit 4
V	=	indicated by "]" preceding label name
?	=	bit 7

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

Using MERLIN With Shift Key Mods

MERLIN supports all hardware shift key modifications. The CONFIGURATION program will establish the modification that you want supported. MERLIN is smart enough to know if the modification actually exists in the Apple you are using and defeats the modification if it is not there. Thus it can be used on another machine without reconfiguration.

Using MERLIN With 8Ø Column Boards

Most, but not all, 8Ø column boards are supported. You may use the VIDEO command to enable the 8Ø column board. To have the board selected upon boot, use the CONFIGURATION program. The Editor's VIDEO Ø or $1Ø command followed by RESET will switch back to the normal Apple screen.

If your board does not support inverse video, then control characters in the source will show up as ordinary capital letters instead of inverse letters as with boards that support inverse. You can use the editor's FIND command to search for particular control characters, verify their presence or absence, or simply switch over to the normal Apple screen.

If your copy of MERLIN has been configured to support an 8Ø column card in slot 3, and there is no card in that slot, MERLIN will recognize this and defeat the 8Ø column provision. There is no need to reconfigure for use on another computer.

MERLIN will NOT support any board that does not recognize the "POKE 36" method of tabbing. As far as we know this only means it will not support older versions of the FULL VIEW 8Ø card.

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 8Ø 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 8Ø column cards, will not do this in EDIT mode and simply produces a CTRL-A in the file line.

The CONFIGURE ASM Program

This program allows you to make several minor modifications to MERLIN's default conditions. It allows you to change the "UPDATE SOURCE" character searched for at the entry to the assembler, the editor's wild card character, and the number of symbol fields printed per line in the symbol table print-out. It also allows you to specify whether you want to have an 8Ø column board supported, and if so, which slot it is in.

You can also specify a hardware shift key modification. Any such modification can be supported. However, if your modification is the type that enables direct input of lower case (as with the VIDEX Keyboard Enhancer) instead of providing a memory location to be tested (as with the "game button 2" modification), then the default at the start of each line will be lower case rather than upper case and CTRL-L will function as a case lock toggle.

CONFIGURE ASM allows you to specify whether you have a lower case adapter. This will affect the condition on boot if you have not elected to have an 8Ø column board selected. It may always be defeated from the editor using the VIDEO command, so this only selects the initial condition.

You may select a number of other options including certain printer options for use by the PRTR command.

Finally, you can save the configured version to another, or the same disk. There is no reason to keep the original version since you can always return to it by reconfiguration.

CONFIGURE ASM can be used to set up the VIDEX ULTRATERM entry defaults. When in the Editor the ULTRATERM mode can be altered by the ESCAPE sequences given in the ULTRATERM manual.

Thus, the following commands give the indicated effects:

```
ESC Ø ..... 4Ø x 24 (same effect as VID $1Ø or 16)
ESC 1 ..... 8Ø x 24 standard character set
ESC 2 ..... 96 x 24
ESC 3 .....16Ø x 24
ESC 4 ..... 8Ø x 24 high quality character set
ESC 5 ..... 8Ø x 32
ESC 6 ..... 8Ø x 48
ESC 7 .....132 x 24
ESC 8 .....128 x 32
```

Note that control V or W commands cannot be issued from the Editor. They can, however, be issued from the Monitor.

If you do not have a lower case adapter then after exiting the ULTRATERM with the ESC Ø command you should use the VID Ø command.

Exit to EXEC mode will return to the default state as set up in the CONFIGURE ASM program and the same is true of a VID 3 command.

Except for the normal 24 x 8Ø 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 4Ø columns before doing this. "CONTROL-I 8ØN" in the PRTR command sometimes overcomes the problem.

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 65Ø2 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.

BAD OPERAND

This occurs if the operand is illegally formed or if a label in the operand is not defined. This also occurs if you "EQU" a label to a zero page value after the label has been used. It may also mean that your operand is longer than 64 characters, or that a comment line exceeds 64 characters. This error will abort assembly.

DUPLICATE SYMBOL

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

MEMORY FULL

This is usually caused by one of four conditions: Incorrect OBJ setting, source code too large, object 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 op-code's request for the value of a label.

BREAK

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

BAD LABEL

This is caused by an unlabeled EQU or MAC, 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).

Special Note - **MEMORY FULL** Errors

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

MEMORY FULL IN LINE: xx. Generated during pass 1 of assembly (line number points to an OBJ instruction). CAUSE: An OBJ was specified that was below MERLIN's HIMEM: (normally $8ØØØ) and also not within Page 3. MERLIN will not allow you to put object code out of this range in order to protect your source file and the system. REMEDY: Remove the OBJ instruction or change it to specify an address within the legal range.

MEMORY FULL IN LINE: xx. Generated during assembly. CAUSE: Too many symbols have been placed into the symbol table, causing it to exceed Applesoft's HIMEM (normally $9853 for the RAM card version and $6F53 for the 48K version). REMEDY: Make the symbol table larger by using the SYM command to lower its beginning address.

ERR: MEMORY FULL. Generated immediately after you type in one line too many. CAUSE: The source code is too large and has exceeded MERLIN's HIMEM (normally $8ØØØ on the RAM card version; $5ØØØ on the 48K version). REMEDY: Raise MERLIN's HIMEM: (see the section on the HImem: command) or break the source file up into smaller sections and bring them in when necessary by using the "PUT" pseudo-op.

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

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

SOURCEROR

Introduction

SOURCEROR is a sophisticated and easy to use disassembler designed as a subsidiary to create MERLIN source files out of binary programs, usually in a matter of minutes. SOURCEROR disassembles SWEET 16 code as well as 65Ø2 code.

The main part of SOURCEROR is called SRCRR.OBJ, but this cannot be run (conveniently) directly, since it may overwrite DOS buffers and crash the system. For this reason, a small program named SOURCEROR is provided. It runs in the input buffer, and does not conflict with any program in memory. This small program simply checks memory size, gets rid of any program such as PLE which would conflict with the main SOURCEROR program, sets MAXFILES 1, then runs SRCRR.OBJ (at $88ØØ-$9AA5).

To minimize the possibility of accident, SRCRR.OBJ has a default location of $4ØØØ and if you BRUN it, it will just return without doing anything. If you try to BRUN it at its designed location of $88ØØ, however, you could be in for big trouble. SOURCEROR assumes the standard Apple screen is being used and will not function with an 8Ø column card.

Using SOURCEROR

1. Load in the program to be disassembled. Although Sourceror will handle programs at any location, the original location for the program is preferable as long as it will not conflict with SOURCEROR and the build up of the source file. When in doubt, load it in at $8ØØ or $8Ø3. Small programs at $6ØØØ and above, or medium sized ones above $4ØØØ will probably be okay at their original locations.

2. BRUN SOURCEROR

3. You will be told that the default address for the source file is $25ØØ. This was selected because it does not conflict with the addresses of most binary programs you may wish to disassemble. Just hit RETURN to accept this default address. Otherwise, specify (in hex) the address you want.

   You may also access a "secret" provision at this point. This is done by typing CTRL-S (for "SWEET") after, or in lieu of the source address. Then you will be asked to specify a (nonstandard) address for the SWEET 16 interpreter. This is intended to facilitate disassembly of programs which use a RAM version of SWEET 16.

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

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

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

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

Commands Used in Disassembly

The disassembly commands are very similar to those used by the disassembler in the 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.

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 65Ø2 mode whenever the SWEET 16 RTN opcode is found.

N (Normal)

This is the same as L, but forces disassembly to start in normal 65Ø2 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.

Any lower case letters appearing in the text string are shown as flashing uppercase letters.

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

Housekeeping Commands

/ (Cancel)

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

As an example, suppose you type T to disassemble some text. You may not know what to expect following the text, so you can just type to L to look at it. Then if the text turns out to be followed by some Hex data (such as $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 $1ØØØ to $1ØFF type 1ØØØR. 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 / before the R.

Q (Quit)

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

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

and the last line is just garbage, type 2344Q. This will cancel the last line, but retain 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 BASIC and lose the disassembled code.

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

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

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

To look at the disassembled source, BRUN MERLIN, or type ASSEM, and load it in.

Dealing with the Finished Source

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

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

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

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

It is important to deal with this situation first as trouble could occur if, for example, the disassembler finds the data AD ØØ 8D. It will disassemble it correctly, as LDA $ØØ8D. The assembler always assembles this code as a zero page instruction, giving the two bytes A5 8D. Occasionally you will find a program that uses this form for a zero page instruction. In that case, you will have to insert a character after the LDA opcode to have it assemble identically to its original form. Often it was data in the first place rather than code, and must be dealt with to get a correct assembly.

The Memory Full Message

When the source file reaches within $6ØØ of the start of SOURCEROR (that is, when it goes beyond $82ØØ) you will see MEMORY FULL and "HIT A KEY" in flashing characters. When you hit a key, SOURCEROR will go directly to the final processing. The reason for the $6ØØ gap is that SOURCEROR needs a certain amount of space for this processing. It is possible (but not likely) that part of SOURCEROR will be overwritten during final processing, but this should not cause problems since the front end of SOURCEROR will not be used again by that point. There is a "secret" override provision at the memory full point. If the key you hit is CTRL-O (for override), then SOURCEROR will return for another command. You can use this to specify the desired ending point. You can also use it to go a little further than SOURCEROR wants you to, and disassemble a few more lines. Obviously, you should not carry this to extremes.

CAUTION: After exiting SOURCEROR, do not try to run it again with a CALL. Instead, run it again from disk. This is because the DOS buffers have been re-established upon exit, and will have partially destroyed SOURCEROR.

The LABELER program

One of the nicest features of the SOURCEROR program is the automatic assignment of labels to all recognizable addresses in the binary file being disassembled. Addresses are recognized by being found in a table which SOURCEROR references during the disassembly process. For example, all JSR $FC58 instructions within a binary file will be listed by SOURCEROR as JSR HOME. This table of address labels may be edited by using the program LABELER.

To use labeler, BRUN LABELER. The program will then mention that SRCRR.OBJ is being loaded into memory, and present the main program menu.

Labeler Commands

Q:QUIT

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

L:LIST

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

D:DELETE LABEL(S)

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

A:ADD LABEL

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

F:FREE SPACE

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

U:UNLOCK SRCRR.OBJ

Before saving a new label table, you will need to UNLOCK the SRCRR.OBJ file. Use this command before Quitting the LABELER program, if you intend to save a new file.

SWEET 16 - INTRODUCTION

by Dick Sedgewick

SWEET 16 is probably the least used and least understood seed in the Apple][.

In exactly the same sense that Integer and Applesoft Basics are languages, SWEET 16 is a language. Compared to the Basics, however, it would be classed as low level with a strong likeness to conventional 65Ø2 Assembly language.

To use SWEET 16, you must learn the language - and to quote WOZ, "The opcode list is short and uncomplicated". "WOZ" (Steve Wozniak), of course is Mr. Apple, and the creator of SWEET 16.

SWEET 16 is ROM based in every Apple][from $F689 to $F7FC. It has its own set of opcodes and instruction sets, and uses the SAVE and RESTORE routines from the Apple Monitor to preserve the 65Ø2 registers when in use, allowing SWEET 16 to be used as a subroutine.

It uses the first 32 locations on zero page to set up its 16 double byte registers, and is therefore not compatible with Applesoft Basic without some additional efforts.

The original article, "SWEET 16: The 65Ø2 Dream Machine", first appeared in Byte Magazine, November 1977 and later in the original "WOZ PAK". The article is included here and again as text material to help understand the use and implementation of SWEET 16.

Examples of the use of SWEET 16 are found in the Programmer's Aid #1, in the Renumber, Append, and Relocate programs. The Programmers Aid Operating Manual contains complete source assembly listings, indexed on page 65.

The demonstration program is written to be introductory and simple, consisting of three parts:

1. Integer Basic Program
2. Machine Language Subroutine
3. SWEET 16 Subroutine

The task of the program will be to move data. Parameters of the move will be entered in the Integer Basic Program.

The "CALL 768" ($3ØØ) at line 12Ø, enters a 65Ø2 machine language subroutine having the single purpose of entering SWEET 16 and subsequently returning to BASIC (addresses $3ØØ, $3Ø1, $3Ø2, and $312 respectively). The SWEET 16 subroutine of course performs the move, and is entered at Hex locations $3Ø3 to $311 (see listing Number 3).

After the move, the screen will display three lines of data, each 8 bytes long, and await entry of a new set of parameters. The three lines of data displayed on the screen are as follows:

Line 1: The first 8 bytes of data starting a $8ØØ, which is the fixed source data to be moved (in this case, the string A$).

Line 2: The first 8 bytes of data starting at the hex address entered as the destination of the move (high order byte only).

Line 3: The first 8 bytes of data starting at $ØØØØ (the first four SWEET 16 registers).

The display of 8 bytes of data was chosen to simplify the illustration of what goes on.

Integer Basic has its own way of recording the string A$. Because the name chosen for the string "A$" is stored in 2 bytes, a total of five housekeeping bytes precede the data entered as A$, leaving only three additional bytes available for display. Integer Basic also adds a housekeeping byte at the end of a string, known as the "string terminator".

Consequently, for convenience purposes of the display, and to see the string terminator as the 8th byte, the string data entered via the keyboard should be limited to two characters, and will appear as the 6th and 7th bytes. Additionally, parameters to be entered include the number of bytes to be moved. A useful range for this demonstration would be 1-8 inclusive, but of course 1-255 will work.

Finally, the starting address of the destination of the move must be entered. Again, for simplicity, only the high-order byte is entered, and the program allows a choice between Decimal 9 and high-order byte of program pointer 1, to avoid unnecessary problems (in this demonstration enter a decimal number between 9 and 144 for a 48K APPLE).

The 8 bytes of data displayed starting at $ØØ will enable one to observe the condition of the SWEET 16 registers after a move has been accomplished, and thereby understand how the SWEET 16 program works.

From the article "SWEET 16: The 65Ø2 Dream Machine", remember that SWEET 16 can establish 16 double byte registers starting at $ØØ. This means that SWEET 16 can use the first 32 addresses on zero page.

The "events" occurring in this demonstration program can be studied in the first four SWEET 16 registers. Therefore, the 8 byte display starting at $ØØØØ is large enough for this purpose.

These four registers are established as RØ, R1, R2, R3:

RØ	$ØØØØ	&	ØØØ1	-SWEET 16 accumulator
R1	$ØØØ2	&	ØØØ3	-Source address
R2	$ØØØ3	&	ØØØ4	-Destination address
R3	$ØØØ4	&	ØØØ5	-Number of bytes to move
.				
.				
.				
R14	$ØØ1C	&	ØØ1D	-Prior result register
R15	$ØØ1E	&	ØØ1F	-SWEET 16 Program counter

Additionally, an examination of registers R14 and R15 will extend an understanding of SWEET 16, as fully explained in the "WOZ" text. Notice that the high order byte of R14, (located at $1D) contains $Ø6, and is the doubled register specification (3X2=$Ø6). R15, the SWEET 16 program counter contains the address of the next operation as it did for each step during execution of the program, which was $Ø312 when execution ended and the 65Ø2 machine code resumed.

To try a sample run, enter the Integer Basic program as shown in Listing #1. Of course, REM statements can be omitted, and line 1Ø is only helpful if the machine code is to be stored on disk. Listing #2 must also be entered starting at $3ØØ.

NOTE: A 65Ø2 disassembly does not look like Listing #3, but the included SOURCEROR disassembler would create a correct disassembly.

p Enter "RUN" and hit RETURN
Enter "12" and hit RETURN (A$ - A$ string data)
Enter "18" and hit RETURN (hi-order byte of destination)

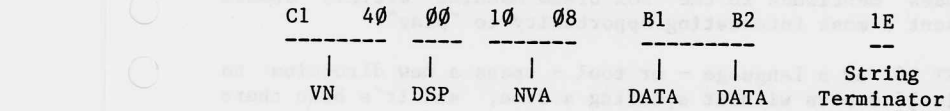
The display should appear as follows:

```
$Ø8ØØ-C1 4Ø ØØ 1Ø Ø8 B1 B2 1E (SOURCE)
$ØAØØ-C1 4Ø ØØ 1Ø Ø8 B1 B2 1E (Dest.)
$ØØØØ-1E ØØ Ø8 Ø8 Ø8 ØA ØØ ØØ (SWEET 16)
```

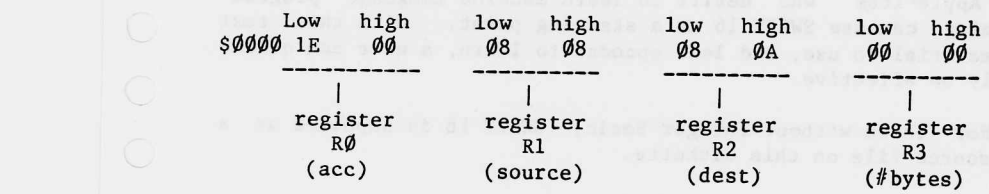
NOTE: The 8 bytes stored at $ØAØØ are identical to the 8 bytes starting at $Ø8ØØ, indicating an accurate move of 8 bytes length has been made. They are moved one byte at a time starting with token C1 and ending with token 1E. If moving less that 8 bytes, the data following the moved data would be whatever existed at those locations before the move.

The bytes have the following significance:

A Token$



The SWEET 16 registers are shown:



The low order byte of R0, the SWEET 16 accumulator, has $1E in it, the last byte moved (the 8th).

The low order byte of the source register R1 started as $00 and was incremented eight times, once for each byte of moved data.

The high order byte of the destination register R2 contains $0A, which was entered as 10 (the variable) and poked into the SWEET 16 code. The low-order byte of R2 was incremented exactly like R1.

Finally, register R3, the register that stores the number of bytes to be moved, has been poked to 8 (the variable B) and decremented eight times as each byte got moved, ending up $0000.

By entering character strings and varying the number of bytes to be moved, the SWEET 16 registers can be observed and the contents predicted.

Working with this demonstration program, and study of the text material will enable you to write SWEET 16 programs that perform additional 16 bit manipulations. The unassigned opcodes mentioned in the "WOZ Dream Machine" article should present a most interesting opportunity to "play".

SWEET 16 as a language - or tool - opens a new direction to Apple][owners without spending a dime, and it's been there all the time.

"Apple-ites" who desire to learn machine language programming can use SWEET 16 as a starting point. With this text material to use, and less opcodes to learn, a user can quickly be effective.

For those without Integer Basic, SWEET 16 is supplied as a source file on this diskette.

Listing #1

```
>List
    1Ø    PRINT "[D]BLOAD SWEET":  REM CTRL D
    2Ø    CALL - 936:  DIM A $ (1Ø)
    3Ø    INPUT  "ENTER STRING A $ " , A $
    4Ø    INPUT  "ENTER # BYTES " , B
    5Ø    IF NOT B THEN 4Ø : REM AT LEAST 1
    6Ø    POKE  778  B : REM POKE LENGTH
    7Ø    INPUT  "ENTER DESTINATION" , A
    8Ø    IF A > PEEK (2Ø3) - 1 THEN 7Ø
    9Ø    IF A < PEEK (2Ø5) + 1 THEN 7Ø
   1ØØ     POKE  776 , A : REM POKE DESTINATION
   11Ø     M = 8 :  GOSUB 16Ø :  REM DISPLAY
   12Ø     CALL  768 : REM GOTO $Ø3ØØ
   13Ø     M = A :  GOSUB 16Ø :  REM DISPLAY
   14Ø     M = O :  GOSUB 16Ø :  REM DISPLAY
   15Ø     PRINT :  PRINT :  GOTO 3Ø
   16Ø     POKE 6Ø , Ø :  POKE  61 , M
   17Ø     CALL  -6Ø5 :  RETURN  :  REM  XAM8  IN MONITOR
```

Listing #2

```
3ØØ:2Ø 89 F6 11 ØØ Ø8 12 ØØ
3Ø8:ØØ 13 ØØ ØØ 41 52 F3 Ø7
31Ø:FB ØØ 6Ø
```

Listing #3

SWEET 16

```
$3ØØ  2Ø  89  F6  JSR   $F689
$3Ø3  11  ØØ  Ø8  SET   R1  source address
$3Ø6  12  ØØ  ØØ  SET   R2  destination address
                   A
$3Ø9  13  ØØ  ØØ  SET   R3  length
                   B
$3ØC  41          LD    @R1
$3ØD  52          ST    @R2
$3ØE  F3          DCR   R3
$3ØF  Ø7          BNZ   $3ØC
$311  ØØ          RTN
$312  6Ø          RTS
```

Data will be poked from the Integer Basic program:

```
"A"    from Line 1ØØ
"B"    from Line 6Ø
```

SWEET 16: A Pseudo 16 Bit Microprocessor

By Steve Wozniak

Description

While writing APPLE BASIC for a 65Ø2 microprocessor, I repeatedly encountered a variant of MURPHY'S LAW. Briefly stated, any routine operating on 16-bit data will require at least twice the code that it should. Programs making extensive use of 16-bit pointers (such as compilers, editors, and assemblers) are included in this category. In my case, even the addition of a few double-byte instructions to the 65Ø2 would have only slightly alleviated the problem. What I really needed was a 65Ø2/RCA 18ØØ hybrid - an abundance of 16-bit registers and excellent pointer capability. My solution was to implement a non-existent (meta) 16-bit processor in software, interpreter style, which I call SWEET 16.

SWEET 16 is based on sixteen 16-bit registers (RØ-15), which are actually 32 memory locations. RØ doubles as the SWEET 16 accumulator (ACC), R15 as the program counter (PC), and R14 as the status register. R13 holds compare instruction results and R12 is the subroutine return stack pointer if SWEET 16 subroutines are used. All other SWEET 16 registers are at the user's unrestricted disposal.

SWEET 16 instructions fall into register and non-register categories. The register ops specify one of the sixteen registers to be used as either a data element or a pointer to data element or a pointer to data in memory, depending on the specific instruction. For example INR R5 uses R5 as data and ST @R7 uses R7 as a pointer to data in memory. Except for the SET instruction, register ops take one byte of code each. The non-register ops are primarily 65Ø2 style branches with the second byte specifying a +/-127 byte displacement relative to the address of the following instruction. Providing that the prior register op result meets a specified branch condition, the displacement is added to the SWEET 16 PC, effecting a branch.

SWEET 16 is intended as a 65Ø2 enhancement package, not a stand-alone processor. A 65Ø2 program switches to SWEET 16 mode with a subroutine call and subsequent code is interpreted as SWEET 16 instructions. The nonregister op RTN returns the user program to 65Ø2 mode after restoring the internal register contents (A, S, Y, P, and S). The following example illustrates how to use SWEET 16.

```
3ØØ  B9 ØØ Ø2          LDA   IN,Y      get a char.
3Ø3  C9 CD             CMP   #"M"      "M" for move
3Ø5  DØ Ø9             BNE   NOMOVE    No. skip move
3Ø7  2Ø 89 F6          JSR   SW16      Yes, call SWEET 16
3ØA  41        MLOOP   LD    @R1       R1 holds source
3ØB  52                ST    @R2       R2 holds dest. addr.
3ØC  F3                DCR   R3        Decr. length
3ØD  Ø7 FB             BNZ   MLOOP     Loop until done
3ØF  ØØ                RTN             Return to 65Ø2 mode.
31Ø  C9 C5     NOMOVE  CMP   #"E"      "E" char?
312  DØ 13             BEQ   EXIT      Yes, exit
314  C8                INY             No, cont.
```

NOTE: Registers A, X, Y, P, and S are not disturbed by SWEET 16.

Instruction Descriptions

The SWEET 16 opcode listing is short and uncomplicated. Excepting relative branch displacements, hand assembly is trivial. All register opcodes are formed by combining two Hex digits, one for the opcode and one to specify a register. For example, opcodes 15 and 45 both specify register R5 while codes 23, 27 and 29 are all ST ops. Most register ops are assigned in complementary pairs to facilitate remembering them. Therefore, LD and ST are opcodes 2N and 3N respectively, while LD @ and ST @ are codes 4N and 5N.

Opcodes Ø to C (Hex) are assigned to the thirteen non-register ops. Except for RTN (opcode Ø), BK (ØA), and RS (ØB), the non register ops are 65Ø2 style branches. The second byte of a branch instruction contains a +/-127 byte displacement value (in two's complement form) relative to the address of the instruction immediately following the branch.

If a specified branch condition is met by the prior register op result, the displacement is added to the PC effecting a branch. Except for BR (Branch always) and BS (Branch to Subroutine), the branch opcodes are assigned in complementary pairs, rendering them easily remembered for hand coding. For example, Branch if Plus and Branch if Minus are opcodes 4 and 5 while Branch if Zero and Branch if NonZero are opcodes 6 and 7.

Sweet 16 Opcode Summary

Register OPS

1n	SET	Rn	Constant (Set)
2n	LD	Rn	(Load)
3n	ST	Rn	(Store)
4n	LD	@Rn	(Load Indirect)
5n	ST	@Rn	(Store Indirect)
6n	LDD	@Rn	(Load Double Indirect)
7n	STD	@Rn	(Store Double Indirect)
8n	POP	@Rn	(Pop Indirect)
9n	STP	@Rn	(Store POP Indirect)
An	ADD	Rn	(Add)
Bn	SUB	Rn	(Sub)
Cn	POPD	@Rn	(Pop Double Indirect)
Dn	CPR	Rn	(Compare)
En	INR	Rn	(Increment)
Fn	DCR	Rn	(Decrement)

Non-register OPS

ØØ	RTN		(Return to 65Ø2 mode)
Ø1	BR	ea	(Branch always)
Ø2	BNC	ea	(Branch if No Carry)
Ø3	BC	ea	(Branch if Carry)
Ø4	BP	ea	(Branch if Plus)
Ø5	BM	ea	(Branch if Minus)
Ø6	BZ	ea	(Branch if Zero)
Ø7	BNZ	ea	(Branch if NonZero)
Ø8	BM1	ea	(Branch if Minus 1)
Ø9	BNM1	ea	(Branch if Not Minus 1)
ØA	BK		(Break)
ØB	RS		(Return from Subroutine)
ØC	BS	ea	(Branch to Subroutine)
ØD			(Unassigned)
ØE			(Unassigned)
ØF			(Unassigned)

Register Instructions

SET

SET Rn,Constant 1n Low High

The 2-byte constant is loaded into Rn (n=Ø to F, Hex) and branch conditions set accordingly. The carry is cleared.

EXAMPLE:

```
15 34 AØ  SET R5, $AØ34    R5 now contains $AØ34
```

LOAD

LD Rn 2n

The ACC (RØ) is loaded from Rn and branch conditions set according to the data transferred. The carry is cleared and contents of Rn are not disturbed.

EXAMPLE:

```
15 34 AØSET R5, $AØ34
25    LD R5              ACC now contains $AØ34
```

STORE

ST Rn 3n

The ACC is stored into Rn and branch conditions set according to the data transferred. The carry is cleared and the ACC contents are not disturbed.

EXAMPLE:

```
25  LD R5                Copy the contents
36  ST R6                of R5 to R6
```

LOAD INDIRECT

LD @Rn 4n

The low-order ACC byte is loaded from the memory location whose address resides in Rn and the high-order ACC byte is cleared. Branch conditions reflect the final ACC contents which will always be positive and never minus 1. The carry is cleared. After the transfer, Rn is incremented by 1.

EXAMPLE:

```
15 34 AØ SET R5, $AØ34
45       LD  @R5        ACC is loaded from memory
                        location $AØ34
                        R5 is incr to $AØ35
```

STORE INDIRECT

ST @Rn 5n

The low-order ACC byte is stored into the memory location whose address resides in Rn. Branch conditions reflect the 2-byte ACC contents. The carry is cleared. After the transfer Rn is incremented by 1.

EXAMPLE:

```
15 34 AØ  SET   R5, $AØ34Load pointers R5, R6 with
16 22 9Ø  SET   R6, $9Ø22$AØ34 and $9Ø22
45        LD    @R5       Move byte from $AØ34 to $9Ø22
56        ST    @R6       Both ptrs are incremented
```

LOAD DOUBLE-BYTE INDIRECT

LDD @Rn 6n

The low order ACC byte is loaded from memory location whose address resides in Rn, and Rn is then incremented by 1. The high order ACC byte is loaded from the memory location whose address resides in the incremented Rn, and Rn is again incremented by 1. Branch conditions reflect the final ACC contents. The carry is cleared.

EXAMPLE:

```
15 34 AØ SET R5, $AØ34    The low-order ACC byte is loaded
65 LDD @R6                from $AØ34, high-order from
                          $AØ35, R5 is incr to $AØ36
```

STORE DOUBLE-BYTE INDIRECT

STD @Rn 7n

The low-order ACC byte is stored into memory location, whose address resides in Rn, and Rn is then incremented by 1. The high-order ACC byte is stored into the memory location whose address resides in the incremented Rn, and Rn is again incremented by 1. Branch conditions reflect the ACC contents which are not disturbed. The carry is cleared.

EXAMPLE:

```
15 34 AØ  SET R5, $AØ34 Load pointers R5, R6
16 22 9Ø  SET R6, $9Ø22 with $AØ34 AND $9Ø22
65        LDD @R5       Move double byte from
76        STD @R        $AØ34-35 TO $9Ø22-23.
                        Both pointers incremented by 2.
```

POP INDIRECT

POP @Rn 8n

The low-order ACC byte is loaded from the memory location whose address resides in Rn after Rn is decremented by 1, and the high order ACC byte is cleared. Branch conditions reflect the final 2-byte ACC contents which will always be positive and never minus one. The carry is cleared. Because Rn is decremented prior to loading the ACC, single byte stacks may be implemented with the ST @Rn and POP @Rn ops (Rn is the stack pointer).

EXAMPLE:

```
15 34 AØ   SET R5, $AØ34     Init stack pointer
1Ø Ø4 ØØ   SET RØ, 4         Load 4 into ACC
55         ST  @R5           PUSH 4 onto stack
1Ø Ø5 ØØ   SET RØ, 5         Load 5 into ACC
55         ST  @R5           Push 5 onto stack
1Ø Ø6 ØØ   SET RØ, 6         Load 6 into ACC
55         ST  @R5           Push 6 onto stack
85         POP @R5           Pop 6 off stack into ACC
85         POP @R5           Pop 5 off stack
85         POP @R5           Pop 4 off stack
```

STORE POP INDIRECT

STP @Rn 9n

The low-order ACC byte is stored into the memory location whose address resides in Rn after Rn is decremented by 1. Branch conditions will reflect the 2-byte ACC contents which are not modified. STP @Rn and POP @Rn are used together to move data blocks beginning at the greatest address and working down. Additionally, single-byte stacks may be implemented with the STP @Rn ops.

EXAMPLE:

```
14 34 AØ   SET R4, $AØ34     Init pointers
15 22 9Ø   SET R5, $9Ø22
84         POP @R4           Move byte from
95         STP @R5           $AØ33 to $9Ø21
84         POP @R4           Move byte from
95         STP @R5           $AØ32 to $9Ø2Ø
```

ADD

ADD Rn An

The contents of Rn are added to the contents of ACC (R) and the low-order 16 bits of the sum restored in ACC. The 17th sum bit becomes the carry and the other branch conditions reflect the final ACC contents.

EXAMPLE:

```
1Ø 34 76   SET RØ, $7634     Init RØ (ACC) and R1
11 27 42   SET R1, $4227
A1         ADD R1              Add R1 (sum=B85B c clear)
AØ         ADD RØ           Double ACC (RØ) to $7ØB6
                             with carry set.
```

SUBTRACT

SUB Rn Bn

The contents of Rn are Subtracted from the ACC contents by performing a two's complement addition:

ACC = ACC + Rn + 1

The low-order 16 bits of the subtraction are restored in the ACC, the 17th sum bit becomes the carry and other branch conditions reflect the final ACC contents. If the 16-bit unsigned ACC contents are greater than or equal to the 16-bit unsigned Rn contents, then the carry is set, otherwise it is cleared. Rn is not disturbed.

EXAMPLE:

```
1Ø 34 76   SET RØ, $7634   Init RØ (ACC)
11 27 42   SET R1, $4227   and R1
B1         SUB R1          Subtract R1
                           (diff=$34ØD with c set)
BØ         SUB RØ           Clears ACC. (RØ)
```

POP DOUBLE-BYTE INDIRECT

POP @Rn Cn

Rn is decremented by 1 and the high-order ACC byte is loaded from the location whose address now resides in Rn. Rn is again decremented by 1 and the low-order ACC byte is loaded from the corresponding memory location. Branch conditions reflect the final ACC contents.

The carry is cleared. Because Rn is decremented prior to loading each of the ACC halves, double-byte stacks may be implemented with the STD @Rn and POPD @Rn ops (Rn is the stack pointer).

EXAMPLE:

```
15 34 AØ  SET R5, $AØ34  Init stack pointer
1Ø 12 AA  SET RØ, $AA12  Load $AA12 into ACC.
75        STD @R5        Push $AA12 onto stack
1Ø 34 BB  SET RØ, $BB34  Load $BB34 into ACC.
75        STD @R5        Push $BB34 onto stack
C5        POPD @R5       Pop $BB34 off stack
C5        POPD @R5       Pop $AA12 off stacy
```

COMPARE

CPR Rn Dn

The ACC (RØ) contents are compared to Rn by performing the 16-bit binary subtraction ACC-Rn and storing the low order 16 difference bits in R13 for subsequent branch tests. If the 16-bit unsigned ACC contents are greater than or equal to the 16-bit unsigned Rn contents, then the carry is set, otherwise it is cleared. No other registers, including ACC and Rn are disturbed.

EXAMPLE:

```
15 34 AØ   SET R5, $AØ34   Pointer to memory
16 BF AØ   SET R6, $AØBF   Limit address
BØ         LOOP1 SUB RØ    Zero data
75         STD @R5         Clear 2 locns
                           Increment R5 by 2
25         LD R5           Compare pointer R5
D6         CPR R6          to limit R6
Ø2 FA      BNC LOOP1       Loop if c clear
```

INCREMENT

INR Rn En

The contents of Rn are incremented by 1. The carry is cleared and other branch conditions reflect the incremented value.

EXAMPLE:

```
15 34 AØ   SET R5, $AØ34   (Pointer)
BØ         SUB RØ          Zero to RØ
55         ST @R5          Clr Locn $AØ34
E5         INR R5          Incr R5 to $AØ36
55         ST @R5          Clrs locn $AØ36
                           (not $AØ35)
```

DECREMENT

DCR Rn Fn

The contents of Rn are decremented by 1. The carry is cleared and other branch conditions reflect the decremented value.

EXAMPLE: (Clear 9 bytes beginning at location AØ34)

```
15 34 AØ   SET R5, $AØ34   Init Pointer
14 Ø9 ØØ   SET R4, 9       Init counter
BØ         SUB RØ          Zero ACC
55         LOOP2 ST @R5    Clear a mem byte
F4         DCR R4          Decrement count
Ø7 FC      BNZ LOOP2       Loop until Zero
```

Non-Register Instructions

RETURN TO 6502 MODE

RTN ØØ

Control is returned to the 65Ø2 and program execution continues at the location immediately following the RTN instruction. The 65Ø2 registers and status conditions are restored to their original contents (prior to entering SWEET 16 mode).

BRANCH ALWAYS

BR ea Ø1 d

An effective address (ea) is calculated by adding the signed displacement byte (d) to the PC. The PC contains the address of the instruction immediately following the BR, or the address of the BR op plus 2. The displacement is a signed two's complement value from -128 to +127. Branch conditions are not changed.

NOTE: The effective address calculation is identical to that for 65Ø2 relative branches. The Hex add & subtract features of the APPLE][monitor may be used to calculate displacements.

```
d = $8Ø   ea = PC + 2 - 128
d = $81   ea = PC + 2 - 127

d = $FF   ea = PC + 2 - 1
d = $ØØ   ea = PC + 2 + Ø
d = $Ø1   ea = PC + 2 + 1

d = $7E   ea = PC + 2 + 126
d = $7F   ea = PC + 2 + 127
```

EXAMPLE:

```
$3ØØ:  Ø1 5Ø  BR $352
```

BRANCH IF NO CARRY

```
BNC ea Ø2  d
```

A branch to the effective address is taken only if the carry is clear, otherwise execution resumes as normal with the next instruction. Branch conditions are not changed.

BRANCH IF CARRY SET

```
BC ea Ø3  d
```

A branch is effected only if the carry is set. Branch conditions are not changed.

BRANCH IF PLUS

BP ea Ø4 d

A branch is effected only if the prior 'result' (or most recently transferred data) was positive. Branch conditions are not changed.

EXAMPLE: (Clear mem from AØ34 to AØ3F)

```
15 34 AØ  SET R5, $AØ34  Init pointer
14 3F AØ  SET R4, $AØ3F  Init limit
BØ        LOOP 3  SUB RØ
55                ST @R5 Clear mem byte
                         ;Increment R5
24                LD R4  Compare limit
D5                CPR R5 to pointer
Ø4 FA           BP LOOP3 Loop until done
```

BRANCH IF MINUS

BM ea Ø5 d

A branch is effected only if prior 'result' was minus (negative, MSB = 1). Branch conditions are not changed.

BRANCH IF ZERO

BZ ea Ø6 d

A branch is effected only if the prior 'result' was zero. Branch conditions are not changed.

BRANCH IF NONZERO

BNZ ea Ø7 d

A branch is effected only if the prior 'result' was non-zero. Branch conditions are not changed.

BRANCH IF MINUS ONE

BM1 ea Ø8 d

A branch is effected only if the prior 'result' was minus one ($FFFF Hex). Branch conditions are not changed.

BRANCH IF NOT MINUS ONE

BNM 1 ea Ø9 d

A branch is effected only if the prior 'result' was not minus 1. Branch conditions are not changed.

BREAK

BK ØA

A 65Ø2 BRK (break) instruction is executed. SWEET 16 may be re-entered non destructively at SW16d after correcting the stack pointer to its value prior to executing the BRK.

RETURN FROM SWEET 16 SUBROUTINE

RS ØB

RS terminates execution of a SWEET 16 subroutine and returns to the SWEET 16 calling program which resumes execution (in SWEET 16 mode). R12, which is the SWEET 16 subroutine return stack pointer, is decremented twice. Branch conditions are not changed.

BRANCH TO SWEET 16 SUBROUTINE

BS ea ØC d

A branch to the effective address (PC + 2 + d) is taken and execution is resumed in SWEET 16 mode. The current PC is pushed onto a 'SWEET 16 subroutine return address' stack whose pointer is R12, and R12 is incremented by 2. The carry is cleared and branch conditions set to indicate the current ACC contents.

EXAMPLE: (Calling a 'memory move' subroutine to move AØ34-AØ3B to 3ØØØ-3ØØ7)

```
15 34 AØ   SET R5, $AØ34   Init pointer 1
14 3B AØ   SET R4, $AØ3B   Init limit 1
16 ØØ 3Ø   SET R6, $3ØØØ   Init pointer 2
ØC 15      BS  MOVE        Call move subrtn
45 MOVE    LD  @R5         Move one
56         ST  @R6         byte
24         LD  R4
D5         CPR R5          Test if done
Ø4 FA      BP  MOVE
ØB         RS              ;Return
```

Theory of Operation

SWEET 16 execution mode begins with a subroutine call to SW16. All 65Ø2 registers are saved at this time, to be restored when a SWEET 16 RTN instruction returns control to the 65Ø2. If you can tolerate indefinite 65Ø2 register contents upon exit, approximately 3Ø usec may be saved by entering at SW16 + 3. Because this might cause an inadvertent switch from Hex to Decimal mode, it is advisable to enter at SW16 the first time through.

After saving the 65Ø2 registers, SWEET 16 initializes its PC (R15) with the subroutine return address off the 65Ø2 stack. SWEET 16's PC points to the location preceding the next instruction to be executed. Following the subroutine call are 1-,2-, and 3-byte SWEET 16 instructions, stored in ascending memory locations like 65Ø2 instructions. The main loop at SW16B repeatedly calls the 'execute instruction' routine to execute it.

Subroutine SW16C increments the PC (R15) and fetches the next opcode, which is either a register op of the form OP REG with OP between 1 and 15 or a non-register op of the form Ø OP with OP between Ø and 13. Assuming a register op, the register specification is doubled to account for the 3 byte SWEET 16 registers and placed in the X-reg for indexing. Then the instruction type is determined. Register ops place the doubled register specification in the high order byte of R 14 indicating the 'prior result register' to subsequent branch instructions. Non-register ops treat the register specification (right-hand half-byte) as their opcode, increment the SWEET 16 PC to point at the displacement byte of branch instructions, load the A-reg with the 'prior result register' index for branch condition testing, and clear the Y-reg.

When is an RTS really a JSR?

Each instruction type has a corresponding subroutine. The subroutine entry points are stored in a table which is directly indexed into by the opcode. By assigning all the entries to a common page, only a single byte of address need be stored per routine. The 65Ø2 indirect jump might have been used as follows to transfer control to the appropriate subroutine.

```
LDA     #ADRH       High-order byte.
STA     IND+1
LDA     OPTBL,X     Low-order byte.
STA     IND
JMP     (IND)
```

To save code, the subroutine entry address (minus 1) is pushed onto the stack, high -order byte first. A 65Ø2 RTS (return from subroutine) is used to pop the address off the stack and into the 65Ø2 PC (after incrementing by 1). The net result is that the desired subroutine is reached by executing a subroutine return instruction!

OPcode Subroutines

The register op routines make use of the 65Ø2 ´zero page indexed by X´ and ´indexed by X indirect´ addressing modes to access the specified registers and indirect data. The ´result´ of most register ops is left in the specified register and can be sensed by subsequent branch instructions, since the register specification is saved in the high-order byte of R14. This specification is changed to indicate RØ (ACC) for ADD and SUB instructions and R13 for the CPR (compare) instruction.

Normally the high-order R14 byte holds the ´prior result register´ index times 2 to account for the 2-byte SWEET 16 registers and the LSB is zero. If ADD, SUB, or CPR instructions generate carries, then this index is incremented, setting the LSB.

The SET instruction increments the PC twice, picking up data bytes in the specified register. In accordance with 65Ø2 convention, the low-order data byte precedes the high-order byte.

Most SWEET 16 non-register ops are relative branches. The corresponding subroutines determine whether or not the 'prior result' meets the specified branch condition and if so, update the SWEET 16 PC by adding the displacement value (-128 to +127 bytes).

The RTN op restores the 65Ø2 register contents, pops the subroutine return stack and jumps indirect through the SWEET 16 PC. This transfers control to the 65Ø2 at the instruction immediately following the RTN instruction.

The BK op actually executes a 65Ø2 break instruction (BRK), transferring control to the interrupt handler.

Any number of subroutine levels may be implemented within SWEET 16 code via the BS (Branch to Subroutine) and RS (Return from Subroutine) instructions. The user must initialize and otherwise not disturb R12 if the SWEET 16 subroutine capability is used since it is utilized as the automatic subroutine return stack pointer.

Memory Allocation

The only storage that must be allocated for SWEET 16 variables are 32 consecutive locations in page zero for the SWEET 16 registers, four locations to save the 65Ø2 register contents, and a few levels of the 65Ø2 subroutine return address stack. If you don't need to preserve the 65Ø2 register contents, delete the SAVE and RESTORE subroutines and the corresponding subroutine calls. This will free the four page zero locations ASAV, XSAV, YSAV, and PSAV.

User Modifications

You may wish to add some of your own instructions to this implementation of SWEET 16. If you use the unassigned op-codes $ØE and $ØF, remember that SWEET 16 treats these as 2-byte instructions. You may wish to handle the break instruction as a SWEET 16 call, saving two bytes of code each time you transfer into SWEET 16 mode. Or you may wish to use the SWEET 16 BK (break) op as a ´CHAROUT´ call in the interrupt handler. You can perform absolute jumps within SWEET 16 by loading the ACC (RØ) with the address you wish to jump to (minus 1) and executing a ST R15 instruction.

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 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: T.APSOFT I, T.APSOFT II, T.APSOFT III and T.APSOFT IV (the 48K version uses T.APSOFT 1 through T.APSOFT 7 instead).

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 that are not normally visible with the CATALOG command 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 list the Applesoft Disassembly

1. BRUN SOURCEROR.FP on your backup copy (see warning above).

2. Boot MERLIN, select the D)rive that contains your backup copy and L)oad APPLESOFT.

3. RAM CARD VERSION: From the editor mode, set SYM to $8ØØØ, enter your PRTR command and ASM)ble the file. The screen should look something like this when you're done:

```
:SYM $8ØØØ
:PRTR 1 "I8ØN"APPLESOFT LISTING"
:ASM
```

4. 48K VERSION: From the editor mode, set HIMEM and SYM to $4EØØ, enter your PRTR command and ASM)ble the file. The screen should look something like this when you're done:

```
:HIMEM:$4EØØ  (<--Note colon)
:SYM $4EØØ
:PRTR 1 "I8ØN"APPLESOFT LISTING"
:ASM
```

In the examples above, the PRTR command will send output to slot 1, initialize the printer interface card with <CTRL I>8ØN" (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, "Ø". 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 do some preliminary checking to make sure everything is OK before printing out the listing. The disk will be accessed a few times, sometimes with long periods between accesses. This is normal. The entire checking process takes about 3.5 minutes.

MERLIN will then begin to print out a completely disassembled and commented listing of Applesoft. It will take 1Ø5 pages (including the symbol tables) and nearly an hour and a half to print out (at a printer rate of 8Ø characters per second).

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 65Ø2 status register.

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

CODE -slang for data or machine language instructions.

CTRL -abbreviation for control or control character.

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

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

DECREMENT -decrease value in constant steps.

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

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

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

EQUATE -establish a variable.

EXPRESSION -actual, implied or symbolic data.

FETCH -retrieve or get.

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

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

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

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

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

INCREMENT -increase value in constant steps.

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

I/O -input/output.

INTERFACE -method of interconnecting peripheral equipment.

INVERT -change to the opposite state.

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

LOOKUP -slang; see table.

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

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

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

MICROPROCESSOR -heart of a microcomputer. (In the Apple, the 65Ø2 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 65Ø2.

OPERAND -data to be operated on by a 65Ø2 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 65Ø2 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 $8Ø.

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

STACK -temporary storage area in RAM used by the 65Ø2 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.

SAMPLE PROGRAMS

The first group of three programs are superb Assembly Language utilities written by Steve Wozniak and Allen Baum, and are still found on the original Integer Basic F4 ROM. They are Supplied in source code format on this diskette for the benefit of Apple][Plus owners who do not have Integer Basic. They may be located at any convenient memory location.

The Floating Point Routines

These are single precision floating point routines that may be interfaced to a BASIC or assembly language program. Information on their use may be found in the source listings themselves.

The Multiply/Divide Routines

These routines are intended to be used as subroutines in assembly language programs providing a four byte multiply or divide result. Brief information on their use is provided in the source listings, and a multiply demo by Dave Garson is included on this diskette.

PRDEC

This is one the most used subroutines in the Integer Basic ROM set. It is called by virtually every routine which requires the output of an integer number in the range Ø-65535. It is easily integrated in any Assembly Language program. To use it, load the accumulator with the high-order byte of [number], load X with the low-byte and call PRDEC. Alternatively, store the high-byte in $F3, the low byte in $F2, and call PRDEC+4.

MSGOUT

This is a subroutine by Andy Hertzfeld to output ASCII strings from an Assembly Language program. If MERLIN INV or FLS Pseudo-ops are used in connection with it, the ORA #$8Ø must be removed, and all normal ASCII must have the high-bit set. Also in the same source file are two simple subroutines to read ASCII and hexadecimal characters input by the user.

UPCON

This utility by Glen Bredon is provided for users who do not have a lowercase video display chip. It will search for source file comments beginning with either "*" or ";", and convert all lower case characters to upper case. Load the source file with MERLIN, then BRUN UPCON, via the ´C´ command in the EXEC mode.

Game Paddle Printer Driver

When the Apple][was first developed, there were no printer interface cards, nor was there really much consideration even given to the need for a printer. Obviously, the folks at Apple computer had a requirement to hard copy their development routines, thus a primitive teletype driver was written by Randy Wiggington and Steve Wozniak to serve their in-house needs. This was subsequently published in the famous "red book" instruction manual, the second for the Apple][. Along came the Disk][, and lo and behold, the driver would not work, since it ignored DOS and set its own I/O hooks. Next the Aldrich brothers took care of this problem, and we were back in business. By this time, of course, there was no desperate need for a game paddle driver; interface cards were developed, and worked well. Nevertheless, some users continued using the game I/O driver, so Dave Garson and Val Golding again modified the drive so that inverse and flashing characters would not upset the printer when doing a catalog, &c.

Concurrently, many new interface cards of all kinds were developed for the Apple: clock cards, 8Ø column cards, ROM cards, &c., until card space is now at a premium. Running a serial printer from the game I/O port is one way in which the user can save both the cost of a printer interface and the slot space it would occupy. Already the teletype driver has been adapted to such printers as Integral Data, Base 2, Heath H-14, and others.

As a last step, Glen Bredon has added a number of improvements to the driver. It can print formatted BASIC listings to any column width, starting with column one, can be output with or without video. The video may be left on even when printing beyond 4Ø columns, something most interface cards can not do. These functions are handled by Basic POKE statements to the flags at the end of the program.

Full documentation and instructions are contained in the source file included on this diskette. Naturally, it is completely compatible with MERLIN, and called with the MERLIN USER command. This is set up when it is first BRUN, which establishes the ampersand hooks, which may also be used from BASIC.

In addition, the source code is well commented, so that it in itself, serves as a tutorial on writing driver routines for different applications, etc.

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 loads itself $9Ø64 and relocates itself to $94AØ. 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 25Ø. Finally, you may specify whether you want the file justified on both sides (rather than just on the left).

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

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

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

CHRGEN 7Ø

CHRGEN 7Ø is a 7Ø-column character generator which is designed specifically to allow the use of MERLIN with a 7Ø column by 24 line display on the Hi-Res screen. Because of the large amount of memory required, CHRGEN 7Ø is available only with the RAM card version of MERLIN.

TV sets do not provide sufficient resolution for use with CHRGEN 7Ø, thus requiring use of a display monitor for satisfactory results.

To use CHRGEN 7Ø, you must first BRUN it from MERLIN's EXEC mode as a DOS command (after a CATALOG). This will reset the source address to $4ØØ1 (above the Hi-Res screen which must be used by CHRGEN 7Ø). This, of course, will delete any source file in memory at the time. Once it has been BRUN, you can invoke it at any time by typing "USER" from the editor.

To exit CHRGEN 7Ø, simply type VID Ø, VID 16, or PR#Ø from the editor. CHRGEN 7Ø is automatically disconnected when you exit the editor to the EXEC Mode. Upon return to the editor, you can reconnect it by typing "USER" again. To permanently remove CHRGEN 7Ø in order to free up the area normally used by long source listings, you will have to BRUN MERLIN again.

To use CHRGEN 7Ø with the editor's PRTR command, just type PRTR 8 "filename", with CHRGEN 7Ø installed in the system.

If the USER vector has been written over by some other USER routine, it can be reset to point to CHRGEN 7Ø either by BRUNing CHRGEN 7Ø again, or by going to the Monitor (use the MON command) and typing in 9ØØG. The latter assumes, of course, that CHRGEN 7Ø is still intact at $9ØØ.

CHRGEN 7Ø includes a version of the FORMATTER program. To implement FORMATTER when CHRGEN 7Ø is connected, just type CTRL-T from the editor's command mode. NOTE: This command may not be accepted unless something has been listed previously.

CHRGEN 7Ø also includes some keyboard macros. Typing the ESCAPE key followed by certain other keys will produce the keyboard macros. These are presently defined for these keys as:

```
     * > " ' # +
     : . 2 7 3 ; X Y D H P @ L S - O C A E
```

The macro table lies at the end of the CHRGEN 7Ø program at $15ØØ and is modifiable. It must end with a $FF.

CAUTION: When CHRGEN 7Ø is up, you must not load any binary source file longer than 88 sectors or it will overwrite the DOS buffers and bomb the system. Text files do not present this danger since they are never allowed to go beyond HIMEM:.

XREF, XREF.XL and STRIP

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

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

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

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

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

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

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

Sample MERLIN Symbol Table Printout:

Symbol table - alphabetical order:

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

Symbol table - numerical order:

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

Sample MERLIN XREF Printout:

Cross referenced symbol table - alphabetical order:

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

Cross referenced symbol table - numerical order:

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

As you can see from the above example (taken from the SWEET 16 source file on the MERLIN diskette), 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. (Your file in memory will now be erased.)

3. Hit <CTRL C> <RETURN> and re-L)oad your file. Initialize your printer with the appropriate PR# or PRTR command (XREF is usually, but not necessarily, a printer oriented command).

4. Type in the appropriate USER command:

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

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

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

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

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

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

NOTE: You may change the USER command as many times as you wish (e.g., from USER 1 to USER 2). The change is not permanent until you enter the ASM command (below).

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

CAUTIONS for the use of XREF

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

Since the program requires assembler output, code in areas with LST OFF will not be processed and labels in those areas will not appear in the table. In particular, it is essential to the proper working of XREF that the LST condition be ON at the end of assembly (since the program also intercepts the regular symbol table output). For the same reason, the 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 1ØØ% reliable.

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

Another thing to look out for when using macros is the fact that labels defined within macro definitions have no global meaning and are therefore not cross-referenced.

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

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

XREF.XL Instructions

XREF.XL is designed to handle files three to four times as large as those handled by XREF. It was originally designed to cross reference the Applesoft Basic source file, which is approximately the largest source file it can process.

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

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

CAUTIONS for the use of XREF.XL:

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

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

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

-Unlike XREF, the setting of HIMEM: does not affect XREF.XL. While building the cross reference table, XREF.XL checks to see if it will fit in the space from the source address (approximately) to the SYM address, if specified, or to $9853 if not. If it is too large, XREF.XL will quit with an OUT OF MEMORY message.

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

Special Instructions for Cross Referencing the Applesoft Source File:

1. Use STRIP on all the Applesoft files and save them on a blank disk.

2. In the file APPLESOFT.S (after stripping it) change the line EXP OFF to EXP ON (or just delete it).

3. Delete the lines containing the SAVOBJ KBD directives and the DO-FIN segment containing the DSK opcode. While this is not necessary, it does avoid having SAVOBJ put in the symbol table.

4. Make sure the last used drive (as shown in the EXEC menu) is the one containing the disk with the stripped files.

5. Enter the desired PRTR command, and then USER 2 or USER 3.

6. Enter ASM and be prepared for a cross reference table approximately 25 pages long.

XREF A and XREF A.XL

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

STRIP

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

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

To use STRIP, follow the following procedure:

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

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

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

4. Q)uit the editor, select the D)rive with MERLIN in it and do a C)atalog. At the COMMAND: prompt, enter: BRUN STRIP. This will remove the comments from your source file in memory.

5. Hit CTRL-C RETURN and enter the Editor. You may now use the XREF and XREF.XL procedures as outlined above.

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

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). Be aware that if you later intend to Read this file using MERLIN's text file reader, you will have to put a "T." in front of the filename in the USER command above; e.g., "T.DUMMY" instead of "DUMMY". Also, you may use a PRTR command instead of USER, if you wish; e.g., PRTR 9 "T.DUMMY"THIS IS A PAGE HEADER"

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 anything 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. Packed files are primarily intended for use with another RWP product, "DOUBLETIME PRINTER", which unpacks the file and "spools" it from disk to the printer.

Video "echoing" means printing on the screen what is sent to the disk. The time it takes to do this can slow PRINTFILER down. (See benchmarking results, next.)

Benchmarking PRINTFILER

Source File: SWEET 16.S (21 sectors)

File Type:	Video :	Sectrs:	Time
Unpacked	Video on :	61	59
Unpacked	Video off:	61	44
Packed	Video on :	38	42
Packed	Video off:	38	3Ø

As you can see from the above, 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:

```
3ØØ:ØØ ØØ  for packed, video off, or.
3ØØ:ØØ 8Ø  for packed, video on, or
3ØØ:8Ø ØØ  for unpacked, video off, or
3ØØ:8Ø 8Ø  for unpacked, video on, or

(normal values are 3ØØ:8Ø ØØ (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 "Ø", 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 "Ø", 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.

CYCLE TIMER

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

5 ,0326 or 5',0326 or 5",0326

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

There are four locations, $9Ø3-$9Ø6, that can be adjusted by the user. You may adjust the position at which the count will be printed, the number (2, 4, or 6) of digits in the accumulated total, the averaging default, and the USR enable flag.

The byte at $9Ø3 contains a number two less than the column in which the cycle count will start printing. The count, however, is always printed after the comment. To have the count printed as far to the right as possible, this byte should be set to the total number of columns minus the number (2, 4, or 6) of digits in the accumulated total minus five. This is presently set to $47 which is appropriate for an 8Ø column printer.

The byte at $904 should contain $80, $40, or 0 depending on whether you want to have 2, 4, or 6 digits in the accumulated total count.

The byte at $905 should contain 0or $80 depending on whether you want the indeterminable added cycles averaged in to the total count or not.

The byte at $9Ø6 should be either Ø or $8Ø depending on whether you want the internal USR routine to be enabled or not. Disabling it simply allows special situtations to be handled. If USR is enabled then the USR opcode without any operand will reset the accumulated total to zero. If the USR opcode has an operand the CYCLE TIMER program will attempt to access the USR routine that was connected when CYCLE TIMER was originally BRUN. Thus, the desired USR routine MUST have been inititalized PRIOR to the BRUN CYCLE TIMER. If $9Ø6 contains $8Ø then a USR opcode is handled as if CYCLE TIMER is not being used.

If your printer allows it, you might try using CYCLE TIMER with the printer set at 96 characters per line rather than the usual 8Ø. This will allow assembly of files ordinarily formatted for the 8Ø columns without having the timer data overflow the lines. Of course, the byte at $9Ø3 must be set accordingly. You must also set the number of columns for the PRTR command to 96. This can be done with the configuration program. It can also be done on a temprorary basis as follows:

1) Type MON from the Editor.

2) Type CØ83 CØ83 RETURN to write enable the language card.

3) Set the byte at $DØØA to the desired number of columns ($6Ø for 96).

4) Type CTRL-Y RETURN to return to MERLIN.

CYCLE TIMER will also work for the extra opcodes in the MERLIN with 65CØ2 mod with the exception of the Rockwell codes - RMB#, SMB#, BBR# and BBS#. These will be ignored by CYCLE TIMER. They are all 5-cycle instructions with the usual possible one or two extra cycles for the branch instructions BBS and BBR.

MERLIN 65C02 ASSEMBLER DOCUMENTATION OVERVIEW

This documentation has been written for those who are already acquainted with 6502 assembly language and wish to learn the operations and addressing modes that have been added to the 65C02.

SECTION ONE

This section will explain how to load Merlin's 65C02 assembly module and also explain operational differences between normal 6502 Merlin and 65C02 Merlin.

SECTION TWO

This section is a brief tutorial on the new instructions.

SECTION THREE

This section is a summary of the new instructions, containing the instruction name, syntax, opcode, number of bytes and number of cycles.

SECTION ONE

LOADING THE 65C02 MODULE INTO MERLIN

There are three files on the Merlin diskette that "convert" Merlin from a normal 6502 assembler to a 65C02 assembler. These are LOAD MODULE 65C02, M65C02, L65C02.

To load the 65C02 module you must BRUN LOAD MODULE 65C02 from Merlin's EXEC mode. This is done simply by pressing C, to catalog the disk and then typing BRUN LOAD MODULE 65C02 when the COMMAND: prompt appears. Once this is done Merlin will be modified to properly assemble 65C02 code. The changes to Merlin will remain in place until the computer is powered down or Merlin is re-booted.

When Merlin is modified to assemble 65C02 code, the Apple monitor's disassembler will properly list the 65C02 codes if the monitor is entered using Merlin's MON command.

There are two limitations when using the 65C02 version of Merlin. The first is that the use of macro names in the opcode column is not allowed. In place of a macro name the ">>>" opcode or the "PMC" opcode must be used. The macro name will then appear in the operand column with the macro's parameters, if any.

The second limitation is that SWEET 16 code will not be assembled. If it is desired to assemble SWEET 16 code while using the 65C02 assembler use macros to define the codes required.

USE OF SOURCEROR WITH 65C02 CODE

Sourceror will properly disassemble 65C02 code if the following procedure is followed. First, LOAD MODULE 65C02 must be BRUN from Merlin's EXEC mode. When the time comes to run SOURCEROR, it must also be BRUN from Merlin's EXEC mode.

SECTION TWO

A DETAILED INTRODUCTION TO THE 65C02

This documentation deals with a new version of the 6502 microprocessor known as the ´65C02´. Although the chip has just been released within the last few months, and as such has yet to find its way into the mainstream of computers, it seems likely that we´ll be hearing more about this item in the upcoming year. The chip can be plugged right into existing Apple computers and is compatible with existing software.

The new 65C02 has had the old 6502 instruction set increased to add a rather large variety of new instructions. Because the chip produced by Rockwell appears to be a superset of all the other new chips, the bulk of this text will assume that this is the chip that is being used. The final portion of this documentation will describe the individual differences that exist between different producer´s versions of the 65C02 chip.

The Rockwell chip has a total of 12 new instructions, and two new addressing modes. In addition a number of addressing modes not normally available to an instruction (such as the immediate mode for the BIT instruction) are now available. There are a total of 59 actual new opcodes. The meaning of all these numbers will become clear shortly.

NEW ADDRESSING MODES

Each 6502 instruction has six possible addressing modes. We will assume, for purposes of simplicity, that a few addressing modes are merely variations of one another. To refresh your memory, a short example is provided here for the LDA (Load Accumulator) instruction.

Addressing Mode	Common Syntax	Hex Coding
1. Absolute	LDA $1234	AD 34 12
Zero Page	LDA $12	A5 12
2. Immediate	LDA #$12	A9 12
3. Absolute,X	LDA $1234,X	BD 34 12
Zero Page,X	LDA $12,X	B5 12
4. Absolute,Y	LDA $1234,Y	B9 34 12
5. (Indirect,X)	LDA ($12,X)	A1 12
6. (Indirect),Y	LDA ($12),Y	B1 12

INDIRECT ADDRESSING

The first of the two new addressing modes is quite easy to explain because it is essentially a variation of an existing instruction. The new mode is "Indirect" addressing. This may sound very familiar because this is similar to the instructions used to access memory locations via a zero page pointer. Usually, though, the Y register is set to zero or some other value, which is then added to the address indicated by the zero page pointer to determine the address we wish to access.

This is fine for accessing a large table of data, but many times we are interested in only one byte of memory, and must then go through the obligatory ´LDY #$00´ (or ´LDX #$00´) to properly condition the Y (or X) register. (See table entries #5 and #6 above).

The new instruction allows us to ignore the contents of the Y register and access the memory location directly. This saves two bytes of code for each reference, since the Y register does not have to be directly loaded. If one wanted to scan a block of memory, such as for a table, this instruction can still be used if you are willing to INC or DEC the zero page pointer accordingly.

Addressing Mode	Common Syntax	Hex Coding
7. Indirect	LDA ($12)	B2 12

This new addressing mode is available for the following instructions:

Instruction & Common Syntax	Hex Coding
ADC ($12)	72 12
AND ($12)	32 12
CMP ($12)	D2 12
EOR ($12)	52 12
LDA ($12)	B2 12
ORA ($12)	12 12
SBC ($12)	F2 12
STA ($12)	92 12

INDEXED ABSOLUTE INDIRECT

The second new addressing mode has a name that was obviously not designed with easy recall in mind. Fortunately, this too is a variation on an existing instruction and as such should be easy to remember. There is currently on the 6502 an indexed indirect addressing, which can be more simply referred to as 'preindexed' addressing. An example is in item #5 in the first table of instructions. Pre-indexing meant that the contents of the X register were added to address of the zero page reference BEFORE using the sum of those numbers to determine which zero page pair to use. For example, the instruction:

```
LDA ($22,X)
```

where the X register held the value '4' would actually access bytes $26,27 to get the final destination address.

This was opposed to indexed indirect, which we can more simply refer to as 'post-indexing'. In post-indexing the value of the Y register is added AFTER the base address is determined. For example, in the instruction:

```
LDA ($22),Y
```

where the Y register holds the value '4', and $22,23 point to location $1000, the memory location accessed would be $1004.

You'll recall also that pre- and post-indexing were limited in their use of the X and Y registers. Pre-indexing could only use the X register and post-indexing only the Y. Before you get too excited in anticipating the possibilities of the new instruction, let me say that not much has changed.

What has changed is that an absolute pre-indexing addressing mode has been added to compliment the indirect JMP. This makes an easy efficient way to create and access JMP tables.

Addressing Mode	Common Syntax	Hex Coding
8. Indexed Absolute Indirect	JMP ($1234,X)	7C 34 12

For example, suppose you had a command interpreter that accepted a command value between 0 and 2. Previously, such an interpreter could be implimented the following ways:

ACTUAL PROGRAM LISTING:

```
                        1  ****************************
                        2  * OLD COMMAND PROCESSOR #1 *
                        3  ****************************
                        4  *
                        5        OBJ $1000
                        6  JMPAD EQU $0A
                        7  * Let's assume that assume that
                           * GETCMD is a subroutine at $4000
                           * that gives us a
                           * number between 0 and 2

1000: 20 00 40          8  ENTRY JSR GETCMD     ; GET VAL FROM
                                                 ; 0 to 2
1003: 0A                9        ASL
1004: AA               10        TAX
1005: BD 13 10         11        LDA CMDAD,X
1008: 85 0A            12        STA JMPAD
100A: E8               13        INX
100B: BD 13 10         14        LDA CMDAD,X
100E: 85 0B            15        STA JMPAD+1
1010: 6C 0A 00         16        JMP (JMPAD)
                       17  *
1013: 00 08            18  CMDAD DA  CMD1
1015: 00 09            19        DA  CMD2
1017: 00 0A            20        DA  CMD3
```

```
                        1  ****************************
                        2  * OLD COMMANND PROCESSOR #2 *
                        3  ****************************
                        4  *
                        5         ORG $1000
1000: 20 00 40          6         JSR GETCMD     ; GET VAL FROM
                                                 ; 0 to 2
1003: AA                7         TAX
1004: BD 0D 10          8         LDA CMDH,X
1007: 48                9         PHA
1008: BD 10 10         10         LDA CMDL,X
100B: 48               11         PHA
100C: 60               12         RTS
                       13  *
100D: 07               14         DFB >CMD1-1    ;COMMAND HIGH
100E: 08               15         DFB >CMD2-1
100F: 09               16         DFB >CMD3-1
                       17  *
1010: FF               18  CMDL   DFB <CMD1-1    ;COMMAND LOW
1011: FF               19         DFB <CMD2-1
1012: FF               20         DFB <CMD3-1

                        1  *************************
                        2  * NEW COMMAND PROCESSOR *
                        3  *************************
                        4  *
1000: 20 00 40          5  ENTRY JSR GETCMD    ;GET VAL FROM
                                               ;0 to 2
1003: 0A                6         ASL          ;MULTIPLY BY 2
1004: AA                7         TAX
1005: 7C 08 10          8         JMP (CMDTBL,X)
                        9  *
1008: 00 08            10  CMDTBL DA CMD1
100A: 00 09            11        DA CMD2
100C: 00 0A            12        DA CMD3
```

Notice how much shorter and straight forward the new instruction makes setting up JMP tables.

NEW 'STANDARD' ADDRESSING MODES

There are a few instructions that have addressing modes that are just "new to them." For example, one of the most exciting ones are INC and DEC.

Previously, any use of INC and DEC was limited to memory locations. In addition, use of the X and Y registers was the only way to maintain a simple loop counter without using a dedicated memory location. The suprise here is that INC and DEC will now work on the accumulator! This is nice because you can now maintain a counter in the accumulator, or even do fudging of flag values, etc. as they are being handled in the accumulator.

Instructions & Common Syntax	Hex Coding
DEC	3A
INC	1A

The BIT instruction also allows some additional addressing modes which may prove useful. The BIT instruction previously only supported absolute addressing. That is to say that a directly referenced memory location was used as the value against which the accumulator was operated on.

Addressing Mode	Common Syntax	Hex Coding
Absolute	BIT $1234	AD 34 12
Zero Page	BIT $12	A5 12

This is useful for testing a memory location for a given bit pattern, but not directly suitable for testing the bit pattern of the accumulator. For many operations, this means you had to rather artificially load some memory location with the value you wanted to compare to the accumulator.

The new 65C02 supports three new addressing modes for the BIT instruction:

1.	Immediate	BIT	#$12	89 12
2.	Absolute,X	BIT	$1234,X	3C 34 12
3.	Zero Page,X	BIT	$12,X	34 12

(It should be noted that BIT immediate affects the N flag as an AND instruction would. Bit 7 and bit 6 of the immediate data does not get transferred to the N and V flags, however.)

NEW INSTRUCTIONS

The new instructions are:

BBR	Branch on Bit Reset (clear)
BBS	Branch on Bit Set
BRA	Branch Always
PHX	Push X onto Stack
PHY	Push Y onto Stack
PLX	Pull X from Stack
PLY	Pull Y from Stack
RMB	Reset (clear) Memory Bit
SMB	Set Memory Bit
STZ	Store Zero
TRB	Test and Reset (clear) Bit
TSB	Test and Set Bit

The following section will explain the function of each of these new instructions.

PHX, PHY, PLX, PLY

The equivalent of these instructions exist for the accumulator but not for the X and Y registers. One of the more common uses for the stack is to save all the registers prior to going into a routine, so that everything can be restored just prior to exiting.

Ordinarily, to save the A, X, and Y registers, we'd have to do something like this:

```
ENTRY  PHA      ; SAVE A
       TXA      ; PUT X IN A
       PHA      ; SAVE IT
       TYA      ; PUT Y IN A
       PHA      ; SAVE IT

WORK   NOP      ; YOUR PROGRAM HERE

DONE   PLA      ; GET Y
       TAY      ; PUT IT BACK
       PLA      ; GET X
       TAX      ; PUT IT BACK
       PLA      ; GET A

EXIT   RTS
```

The problem is even further complicated in programs where you need not only to save the registers but to use the accumulator for other purposes also at the same time.

With the new 65C02, this could all be resolved with:

```
ENTRY  PHX      ; SAVE X
       PHY      ; SAVE Y
       PHA      ; SAVE A, BUT LEAVE IT ON TOP

WORK   NOP      ; THE PROGRAM HERE

DONE   PLA      ; GET A
       PLY      ; GET Y
       PLX      ; GET X

EXIT   RTS
```

Addressing Mode	Common Syntax	Hex Coding
Implied only	PHX	DA
	PHY	5A
	PLX	FA
	PLY	7A

BRA (Branch Always)

This is one of those instructions that will thrill writers of relocatable code. In Assembly Lines: The Book, it is discussed how to write code that is location independent, and one of the techniques was the use of a forced branch instruction, such as:

```
CLC             ; CLEAR CARRY
BCC   LABEL     ; ALWAYS
```

Unfortunately, this means we must force some flag of the status register, which may not be convenient at the time. In addition, the process takes up an extra byte on most occasions.

Branch Always alleviates both these problems by always branching to the desired address, subject of course to the usual limitations of plus or minus 128 bytes as the maximum branching distance.

It is worth mentioning in the interest of programming style, that many people indiscriminately use a JMP to go back to the top of a loop when a branch instruction would do the trick, and add one more limitation to the final code in the process. Hopefully, this new branch instruction will encourage people to make their code more location independent.

Addressing Mode	Common Syntax	Hex Coding
Relative only	BRA LABEL1	80 12

STZ (Store Zero)

This instruction is used for zeroing out memory bytes without changing the contents of any of the registers.

Many times it is necessary to set a number of internal program registers to zero before proceeding with the routine. This is especially needed in mathematical routines such as multiplication and division.

Ordinarily, this is done by loading the accumulator with zero, and then storing that value in the appropriate memory locations. This is easy to do when you have to load the A, X or Y registers with zero anyway. The problem is that on occasion, the ONLY reason one of the registers is loaded with zero is because of the need to zero a memory location.

Store Zero allows us to zero out any memory byte without concern to current register contents. The same variety of addressing modes usually available to a STA, STX or STY instruction is not available with a STZ though.

Addressing Mode	Common Syntax	Hex Coding
Absolute	STZ $1234	9C 34 12
Zero Page	STZ $12	64 12
Absolute,X	STZ $1234,X	9E 34 12
Zero Page,X	STZ $12,X	74 12

SMB, RMB (Set/Reset Memory Bit) [Rockwell chip only]

This pair of instructions will allow you to set or clear a given bit of a byte on the zero page. Previously, this would have required three seperate instructions to achieve the same result. For example:

```
    LDA MEMORY      ; LOAD VALUE FROM MEMORY
    AND #$7F        ; %0111 1111 IS PATTERN
                    ; NEEDED TO CLR BIT 7
    STA MEMORY      ; PUT IT BACK
```

With the new instruction, we can accomplish the same thing with:

```
    RMB7  MEMORY    ; RESET (CLR) BIT 7 OF MEMORY
```

or put it back with:

```
    SMB7 MEMORY     ; SET BIT 7 OF MEMORY
```

Two interesting things to note here. The first is that for some reason they use the term "RESET" instead of clear to indicate the zeroing of a given bit.

The second item is that we now have four-character instruction codes (mnemonics), it seems this new species of instruction has arrived.

Addressing Mode	Common Syntax	Hex Coding
Zero Page, Bit 0	SMB0 $12	87 12
.	.	.
.	.	.
Zero Page, Bit 7	SMB7 $12	F7 12
Zero Page, Bit 0	RMB0 $12	07 12
.	.	.
.	.	.
Zero Page, Bit 7	RMB7 $12	77 12

These instructions are limited to zero page addressing.

BBS, BBR (Branch on Bit Set/Reset) [Rockwell only]

These two new branch instructions make it possible to test any bit of a zero page location, and then branch depending on its condition. This instruction would be very useful for testing flags in programs that need to pack flag-type data into as few bytes as possible. By transferring I/O device registers, etc. to zero page, it is also possible to directly test bits in these registers for status bit conditions.

These instructions are very similar to the Bit Set and Reset instructions just discussed in there appearance and use. The difference is, of course, that we are testing bit status, rather than changing it.

Addressing Mode	Common Syntax	Hex Coding
Zero Page, Bit 0	BBS0 $12,LABEL	8F 12 FF
.	.	.
.	.	.
Zero Page, Bit 7	BBS7 $12,LABEL	FF 12 FF
Zero Page, Bit 0	BBR0 $12,LABEL	0F 12 FF
.	.	.
.	.	.
Zero Page, Bit 7	BBR7 $12,LABEL	7F 12 FF

One of the first applications that springs to mind is the testing of whether a number is odd or even. Previously, this had to be done with a LSR or ROR instruction, followed by a test of the carry flag, such as:

```
LDA MEMORY      ;LOAD A WITH VALUE
LSR             ;SHIFT BIT 0 INTO CARRY
BCS ODD         ;SET IF ODD
BCC EVEN        ;CLR IF EVEN
```

The equivalent can now be done without effecting the carry flag or the accumulator:

```
BBR0 MEMORY,EVEN  ;BRANCH IF BIT 0=0 = EVEN
BBS0 MEMORY,ODD   ;BRANCH IF BIT 0=1 = ODD
```

This could be useful also in creating drivers for the new Apple //e 80 column extended memory board since this card uses one bank of memory or another depending on whether the screen column position is odd or even.

TSB, TRB (Test and Set/Reset Bit)

These instructions are like a combination of the BIT and AND/ORA instructions of the old 6502.

The instructions seem primarily designed for controlling I/O devices, but may have other interesting applications as things develop.

The action of these two instructions is to use a mask stored in the accumulator to condition a memory location. The mask in the accumulator is unaltered, but the Z flag of the status register is conditioned depending on the memory contents prior to the operation. The Z flag is conditioned based on the result of the accumulator ANDed with the byte in memory like the BIT instruction.

For example, to set both bits 0 and 7 of a memory location, we could use the following set of instructions:

```
LDA #$81          ;%1000 0001 = MASK PATTERN
TSB MEM1          ;SET BITS 0,7 OF MEMORY
BNE PRSET         ;ONE OR THESE WAS 'ON' ALREADY
BEQ PRCLR         ;NONE OF THESE WERE 'ON'
```

This would clear the bits:

```
LDA #$81          ;%1000 001 = MASK PATTERN
TRB MEM2          ;CLR BIT 0,7 OF MEMORY
BNE PRSET         ;ONE OF THESE WAS 'ON' ALREADY
BEQ PRCLR         ;NONE OF THESE WERE 'ON'
```

The addressing modes allowed for these instructions are as follows:

Addressing Mode	Common Syntax	Hex Coding
Absolute	TRB $1234	1C 34 12
Zero Page	TRB $12	14 12
Absolute	TSB $1234	0C 34 12
Zero Page	TSB $12	04 12

OTHER DIFFERENCES

There are a number of other differences in the chips, most notably the power consumption. The power use of the 65C02 is one tenth that of the 6502 so the chip runs considerably cooler, not to mention the possibilities opened for portable computers, terminals, etc.

One point of interest is that the old 6502 bug of the indirect jump problem has been fixed. If you're not aware of it, the 6502 has a well-documented problem with indirect jumps that use a pair of bytes that straddle a page boundary.

For example, consider these two instructions:

Instruction	Pointers Wanted	Pointers Used
JMP ($380)	$380,381	$380,381
JMP ($3FF)	$3FF,400	$3FF,300

Notice that in the last instance, the pointers used are NOT those anticipated. This is because the high byte of the pointer address does not get properly incremented on the standard 6502.

This problem has been fixed on the 65C02. The only possible problem here is "clever" protection schemes that may have used this bug to throw off people trying to decode the protection method. Otherwise, this should not present any problems to existing software.

Are there any problems to be anticipated? In theory, no. The new 65C02 is pin-for-pin compatible with the old one, and also upward compatible in terms of software. ANY software for the Apple, PET, Atari or other 6502 based microcomputers SHOULD work without problems with the new chip. Unfortunately, in reality there can be problems.

The first big problem concerns internal microprocessor timing on the Apple II and II+ computers. From the available information it seems that the Apple II and II+ do not handle the CPU clock cycles in the same way that the IIe does.

On the surface, the 65C02 should directly replace the 6502. However, because the 65C02 is a faster chip, data is not available for as long, and bits can get lost. What this means for now is that the 65C02 can only be used on the Apple IIe and Apple III machines. None of the manufacturers at this time produce a chip that works in an Apple II or II+.

There is also the possibility of problems with some existing software. A very small percentage of software may be using (it's not really known) undocumented bugs or "features" of the old 6502 chip, and these might not function as anticipated.

Where did all of the "new" opcodes come from? To answer this, consider how the instructions we use now are structured. The 6502 operates by scanning memory and performing specific operations depending on the values that it finds in each memory location. You would expect a total of 256 possible opcodes then. As it happens, all 256 possible values are not used. It is this group of "unused" opcodes that allows for the "new" instructions, and also creates the possibility of difficulties for a small percentage of existing programs.

As an example of how difficulties may arise let's look at the $FF opcode from the 6502. The code $FF on a 6502 is labeled as an alternate NOP. As such, some existing programs may use $FF instead of the usual $EA to implement a NOP. Although rarely documented, the previously "unused" values, such as $FF, will cause certain things to happen, much the same way that a legal value would. As it happens, the $FF is one of the codes that has been converted to a new function in the 65C02, namely BBS7 (Branch on Bit 7 Set).

Other unused codes often have combination effects, usually of little use, such as loading the accumuator and decrementing a register at the same time. Their main application is similar to the indirect jump problem, i.e. creating code that cannot be casually interpreted. If these instructions have been used in existing software though, problems could arise with the 65C02.

At this writing, the Rockwell chip seems to have the largest set of instructions of the three varieties available. The GTE and NCR chips lack the bit manipulation instructions (BBS, BBR, SMB, and RMB) , but are otherwise identical.

SECTION THREE

SUMMARY OF NEW INSTRUCTIONS

THE FOLLOWING 6502 INSTRUCTIONS GET NEW ADDRESSING MODES

ADC
AND
BIT
CMP
DEC
EOR
INC
JMP
LDA
ORA
STA

The following instructions get ZERO PAGE INDIRECT ADDRESSING

ADC
AND
CMP
EOR
LDA
ORA
STA

ACCUMULATOR ADDRESSING

DEC
INC

ABSOLUTE Pre-Indexed Indirect Addressing

JMP (addr16,X) Usefull for jump tables.

The BIT instruction has the following additional addressing modes.

```
BIT imm.
BIT ZP,X
BIT ABS,X
```

The following new instructions have been added.

```
BBR  Branch on Bit Reset  [ROCKWELL CHIP ONLY]
BBS  Branch on Bit Set    [ROCKWELL CHIP ONLY]
BRA  Branch Always
PHX  Push X Register on Stack
PHY  Push Y Register on Stack
PLX  Pull X Register from Stack
PLY  Pull Y Register from Stack
RMB  Reset Memory Bit           [ROCKWELL CHIP ONLY]
SMB  Set Memory Bit             [ROCKWELL CHIP ONLY]
STZ  Store Zero
TRB  Test and Reset Bits
TSB  Test and Set Bits
```

-> BRA Branch always

MODE	SYNTAX	OPCODE	BYTES	CYCLES
relative	BRA offset	$80	2	3

DESCRIPTION: BRA executes an unconditional branch. This is a relative branch and as such is limited to +129/-126 offset from the address of the instruction.
USES: This additional branch is useful in writing position independent code. It executes in the same number of cycles (3) as an absolute JMP, but requires only two bytes. This results in tighter code if the relative offset is close enough.

-> PHX Push X Register on Stack

MODE	SYNTAX	OPCODE	BYTES	CYCLES
Implied	PHX	$DA	1	3

DESCRIPTION: Save the contents of the X register on the stack.

-> PHY Push Y Register on Stack

MODE	SYNTAX	OPCODE	BYTES	CYCLES
Implied	PHY	$5A	1	3

DESCRIPTION: Save the contents of the Y register on the stack.

-> PLX Pull X Register from Stack

MODE	SYNTAX	OPCODE	BYTES	CYCLES
Implied	PLX	$FA	1	4

DESCRIPTION: Load the X register from the stack.

-> PLY Pull Y Register from Stack

MODE	SYNTAX	OPCODE	BYTES	CYCLES
Implied	PLY	$7A	1	4

NOTE: PLX and PLY affect the processor status register as PLA.

-> RMB Reset Memory Bit [Rockwell Chip Only]

MODE	SYNTAX	OPCODE	BYTES	CYCLES
Zero Page	RMB0 add8	$07	2	5
Zero Page	RMB1 add8	$17	2	5
Zero Page	RMB2 add8	$27	2	5
Zero Page	RMB3 add8	$37	2	5
Zero Page	RMB4 add8	$47	2	5
Zero Page	RMB5 add8	$57	2	5
Zero Page	RMB6 add8	$67	2	5
Zero Page	RMB7 add8	$77	2	5

DESCRIPTION: The specified memory bit is set to zero. This instruction does not affect the status register.

-> SMB Set Memory Bit [Rockwell Chip Only]

MODE	SYNTAX	OPCODE	BYTES	CYCLES
Zero Page	SMB0 add8	$87	2	5
Zero Page	SMB1 add8	$97	2	5
Zero Page	SMB2 add8	$A7	2	5
Zero Page	SMB3 add8	$B7	2	5
Zero Page	SMB4 add8	$C7	2	5
Zero Page	SMB5 add8	$D7	2	5
Zero Page	SMB6 add8	$E7	2	5
Zero Page	SMB7 add8	$F7	2	5

DESCRIPTION: The specified memory bit is set to one. This instruction does not affect the status register.

-> STZ Store Zero

MODE	SYNTAX	OPCODE	BYTES	CYCLES
Absolute	STZ addr16	$9C	3	4
Zero Page	STZ addr8	$64	2	3
Zero Page,X	STZ addr8,X	$74	2	4
Abs,X	STZ addr16,X	$9E	3	5

DESCRIPTION: Store a zero in the specified memory location. This instruction does not affect the status register.

-> TRB Test and Reset Bits

MODE	SYNTAX	OPCODE	BYTES	CYCLES
Absolute	TRB addr16	$1C	3	6
Zero Page	TRB addr8	$14	2	5

Description: Reset the bits of the specified memory location based on the mask in the accumulator. The accumulator is left unaltered. The Z flag is conditioned based on the result A AND M before the operation takes place (i.e. BIT instruction).

-> TSB Test and Set Bits

MODE	SYNTAX	OPCODE	BYTES	CYCLES
Absolute	TSB Addr16	$0C	3	6
Zero Page	TSB Addr8	$04	2	5

Description: set the bits of the specified memory location based on the mask in the accumulator. The accumulator is left unaltered. The Z flag is conditioned based on the result A AND M before the operation takes place (i.e. BIT instruction).

-> BBRn Branch on Bit n Reset. [Rockwell Chip Only]

MODE	SYNTAX	OPCODE	BYTES	CYCLES
RELATIVE	BBR0 addr8,offset	$0F	3	5+n
RELATIVE	BBR1 addr8,offset	$1F	3	5+n
RELATIVE	BBR2 addr8,offset	$2F	3	5+n
RELATIVE	BBR3 addr8,offset	$3F	3	5+n
RELATIVE	BBR4 addr8,offset	$4F	3	5+n
RELATIVE	BBR5 addr8,offset	$5F	3	5+n
RELATIVE	BBR6 addr8,offset	$6F	3	5+n
RELATIVE	BBR7 addr8,offset	$7F	3	5+n

n = 0 if branch does not occur
n = 1 if branch occurs to same page
n = 2 if branch occurs to different page

Description: This instruction causes a Branch to take place if the selected bit of the specified zero page location is reset (=0). No flags are affected.

-> BBSn Branch on Bit n Set. [Rockwell Chip Only]

MODE	SYNTAX	OPCODE	BYTES	CYCLES
RELATIVE	BBS0 addr8,offset	$8F	3	5+n
.	.	.	.	.
.	.	.	.	.
RELATIVE	BBS7 addr8,offset	$FF	3	5+n

n = 0 if branch does not occur
n = 1 if branch occurs to same page
n = 2 if branch occurs to different page

Description: This instruction causes a Branch to take place if the selected bit of the specified zero page location is set (=1). No flags are affected.

C

D

Roger Wagner™
PUBLISHING, INC.

P.O. BOX 582 ● SANTEE, CA 92071 ● FOR TECHNICAL SUPPORT OR CUSTOMER SERVICE, CALL 619/562-3670

SOFTWARE REGISTRATION FORM

THANK YOU for purchasing our product. PLEASE complete this form & mail to Roger Wagner Publishing, Inc.

PROGRAM NAME: _____

CUSTOMER NAME: _____

STREET ADDRESS: _____

CITY: _____ STATE: _____ ZIP: _____

TELEPHONE: _____ AREA CODE: _____

DATE PURCHASED: _____ PURCHASE PRICE: _____

WHERE PURCHASED: _____

STREET ADDRESS: _____

CITY: _____ STATE: _____ ZIP: _____

TELEPHONE: _____ AREA CODE: _____

HOW DID YOU SELECT PROGRAM? ☐ Salesperson/Demo? ☐ Advertisements?
☐ Brand Name (RWP) Recognition? ☐ Friend's Recommendation?

What is your overall opinion of the product? _____

_____

How can we improve this product? _____

_____

DO YOU OWN ANY OTHER RWP PRODUCTS? If yes, specify: _____

_____

DO YOU DO ANY PROGRAMMING YOURSELF? ☐ Yes ☐ No

If yes, which? ☐ Professional ☐ Personal Use ☐ Both

What types of programs do you write? _____

_____

WHAT TYPES OF SOFTWARE DO YOU WANT TO PURCHASE IN THE FUTURE?

☐ BUSINESS: _____ ☐ UTILITIES: _____

_____ _____

_____ _____

_____ _____

☐ GAMES: _____ ☐ EDUCATIONAL: _____

_____ _____

_____ _____

_____ _____

WHICH COMPUTER MAGAZINES DO YOU READ? _____

_____

DO YOU HAVE CHILDREN WHO USE YOUR APPLE? ☐ Yes ☐ No

If yes, what are their ages? _____

HOW LONG HAVE YOU OWNED YOUR COMPUTER SYSTEM? _____

WOULD YOU LIKE TO RECEIVE THE RWP PRODUCT GUIDE? _____

PLACE
STAMP
HERE

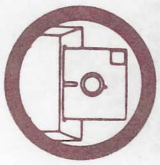
Roger Wagner™
PUBLISHING, INC.

P.O. Box 582
Santee, California 92071


Roger Wagner
PUBLISHING, INC.
©1983
MERLIN
6502 Macro Assembler
For Apple II/II+ and IIe - DOS 3.3
MASTER: COPY BEFORE USING
FP SOURCEROR ON REVERSE SIDE
RogerWagner™
PUBLISHING, INC.

For extended media life—
here's how to take care of your flexible disk

Precision surface.
No fingers, please!

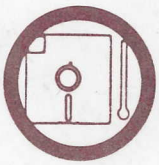


For your disk's sake
(and the system's, too)
insert disk carefully.

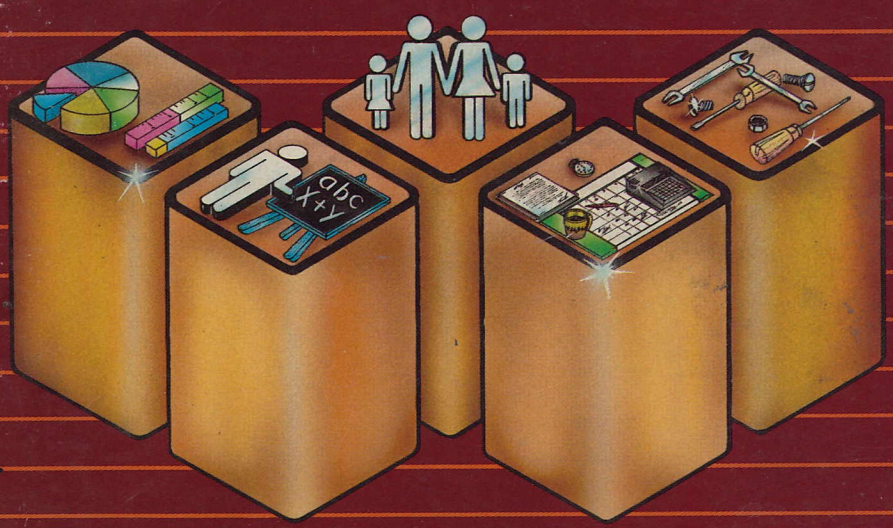
Magnetic fields erase.
Keep them far away.

Keep it safe—
in the envelope
when not in use.

Bending and folding
may damage.
Handle with care.



Keep disks comfortable
Store at: 10° to 50° C.
50° to 125° F.



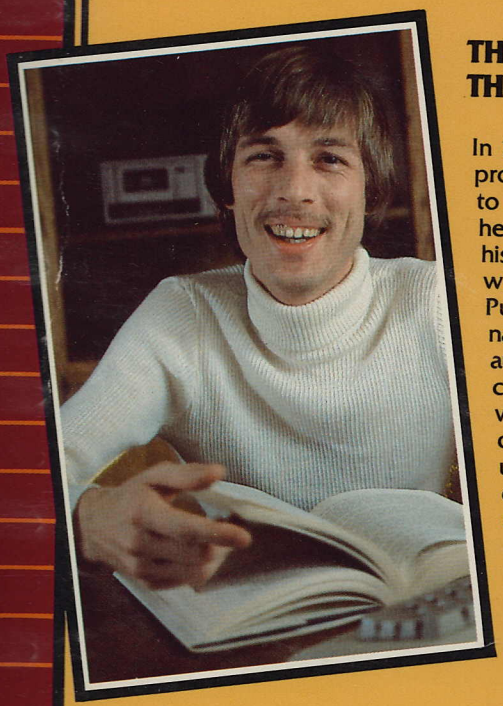
Roger Wagner™
PUBLISHING, INC.

Roger Wagner™
PUBLISHING, INC.

GRAPHICS: DOUG WESTERKAMP & ASSOCIATES
COVER ILLUSTRATION: SCOTTY ZIEGLER
PRINTING: LOYND COMMUNICATIONS

EL CAJON, CALIFORNIA

About the Publisher...



THE MAN BEHIND THE NAME

In 1978, Roger Wagner began writing programs for two reasons; he wanted to make programming easier, and he wanted to learn more about his computer. He accomplished both while founding Roger Wagner Publishing, and he has achieved national acclaim as a programmer, author, columnist and speaker. His contributions to the microcomputer world demonstrate his interest and dedication to supplying the home user with software that both operates and educates. As Softalk magazine noted, "His programs reflect a concern that the user get more than utility — he should also gain knowledge — from use of the software." Thousands of home users have benefitted from this simple concept.

Today, Roger Wagner works with some of the most talented authors in the world during program development to ensure that this tradition continues. Typical state of the art features include unprotected and listable software, instruction manuals with tutorials written in plain English, hard disk compatibility, easy to use menus, online help notes, free bonus software and more — all designed to give you the advantage by increasing your personal productivity. Roger Wagner software makes it easy to get the most out of your computer.

Roger Wagner™
PUBLISHING, INC.

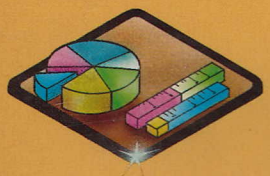
Roger Wagner™
PUBLISHING, INC.

With exciting software that's fun to use and instructions that are easy to understand, we make using your computer an activity shared by the whole family.

Expand your knowledge of a specific subject and extend your understanding of your computer in the process. Learning will be an enjoyable experience for the whole family.

From novice to professional, from BASIC to Assembly Language, special software tools designed to make writing your own programs easy and to release the true power of your computer.

Manage all your information needs and quickly get the information you want, when you want it. Keep your input to a minimum and output to a maximum.



Guarantee your investment in the future and make your money matter. Chart your course for tomorrow with software that pays for itself every day.

MERLIN™

The Macro Assembler For The Apple II Family

Roger Wagner™
PUBLISHING, INC.

Roger Wagner

Roger Wagner™
PUBLISHING, INC.