A.P.P.L.E.

PRESENTS:

Big Mac

MACRO-ASSEMBLER/TED

Apple PugetSound Program Library Exchange

Ted Mikula

DISCLAIMER

This manual and the accompanying diskette are available only to members of Apple Pugetsound Program Library Exchange.

A.P.P.L.E. PRODUCTS ARE SOLD "AS IS". A.P.P.L.E. DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR USE.

SECTION I QUICK REFERENCE COMMAND SUMMARY

Command	Description	Page
	EXEC MODE	
C: CATALOG	Display catalog and allow DOS commands	9
L: LOAD	Load a source file from disk	9
R: READ	Read a text file from disk	LC
S: SAVE	Save a source file to disk	9
W: WRITE	Write a text file to disk	LC
A: APPEND	Load a source file at end of file in memory	9
D: DRIVE	Toggle from drive 1 to drive 2	10
E: EDIT	Enter edit/asm mode	10
Z: ZERO TABS	Enter edit/asm mode with tabs set to 0	10
O: OBJ SAVE	Save object code after successful assembly	10
Q: QUIT	Exit to BASIC	10
	EDITOR	
	Command Mode	
H*Imem:*	Sets upper limit for source file	11
NEW	Deletes present source, resets Himem:	11
PR*#*	Same function as BASIC PR*#*	11
USER	Executes user routine at $3F5	11
TABS	Sets tab stops for editor listing	12
LEN*gth*	Returns number of bytes in source file	12
Where	Returns memory address of specified line number	12
MON*itor*	Exits to monitor. Return with Ctrl Y	12
TR*unc*ON	Omits comments prefixed " ;" &ASCII, HEX obj code	12
TR*unc*OF	Reset truncate flag to default	12
STRIP	Strip comments prefixed "*" or " ;" from source	LC
Quit	Exit to EXEC mode	12
SYM	Establishes user symbol table area	LC
ASM	Commences assembly	13
Delete	Delete line number, range, or range list	13
Replace	As above, then falls into Insert mode	13
List	List source file with line numbers	13
Print	List source without line numbers	13
/	Continue List from last line number	14
Find	Find d-string specified	14
Change	Replace d-string1 with d-string 2	14
COPY	Copy line number range to above specified line	14
MOVE	As above, but deletes original lines	14
Edit	Edit line number or range specified	14
	Add/insert mode	
Add	Enter text entry mode	15
Insert	Enter text entry mode just above specified line	15
Ctrl L	Case toggle: select opposite case	15
Ctrl O	Enter non-keyboard characters	LC
Ctrl X	Exit text entry mode	15
	Edit mode	
Ctrl I	Insert character(s)	16
Ctrl O	Insert Control character	16
Ctrl D	Delete character(s)	16
Ctrl F	Find next occurrence of character after Ctrl F	16
Ctrl B	Place cursor at beginning of line	16
Ctrl N	Place cursor at end of line	16
Ctrl L	Change case of character under cursor	16
Ctrl R	Restore line to original form	16
Ctrl C	Exit Edit mode	16
Ctrl Q	Enter line up to cursor position	16
[return]	Enter entire line as it appears	16

LC = Command available on Language Card version only.

ASSEMBLES AT $8000
$6=4C 06 08

Big Mac

MACRO-ASSEMBLER/TED

by Glen E. Bredon

BIG MAC
TABLE OF CONTENTS

SECTION I QUICK REFERENCE COMMAND SUMMARY

Command	Description	Page
	EXEC MODE	
C: CATALOG	Display catalog and allow DOS commands	9
L: LOAD	Load a source file from disk	9
R: READ	Read a text file from disk	LC
S: SAVE	Save a source file to disk	9
W: WRITE	Write a text file to disk	LC
A: APPEND	Load a source file at end of file in memory	9
D: DRIVE	Toggle from drive 1 to drive 2	10
E: EDIT	Enter edit/asm mode	10
Z: ZERO TABS	Enter edit/asm mode with tabs set to 0	10
O: OBJ SAVE	Save object code after successful assembly	10
Q: QUIT	Exit to BASIC	10
	EDITOR	
	Command Mode	
HI*mem:*	Sets upper limit for source file	11
NEW	Deletes present source, resets Himem:	11
PR*#*	Same function as BASIC PR*#*	11
USER	Executes user routine at $3F5	11
TABS	Sets tab stops for editor listing	12
LEN*gth*	Returns number of bytes in source file	12
Where	Returns memory address of specified line number	12
MON*itor*	Exits to monitor. Return with Ctrl Y	12
TR*unc*ON	Omits comments prefixed " ;" &ASCII, HEX obj code	12
TR*unc*OF	Reset truncate flag to default	12
STRIP	Strip comments prefixed "*" or " ;" from source	LC
Quit	Exit to EXEC mode	12
SYM	Establishes user symbol table area	LC
ASM	Commences assembly	13
Delete	Delete line number, range, or range list	13
Replace	As above, then falls into Insert mode	13
List	List source file with line numbers	13
Print	List source without line numbers	13
/	Continue List from last line number	14
Find	Find d-string specified	14
Change	Replace d-string1 with d-string 2	14
COPY	Copy line number range to above specified line	14
MOVE	As above, but deletes original lines	14
Edit	Edit line number or range specified	14
	Add/insert mode	
Add	Enter text entry mode	15
Insert	Enter text entry mode just above specified line	15
Ctrl L	Case toggle: select opposite case	15
Ctrl O	Enter non-keyboard characters	LC
Ctrl X	Exit text entry mode	15
	Edit mode	
Ctrl I	Insert character(s)	16
Ctrl O	Insert Control character	16
Ctrl D	Delete character(s)	16
Ctrl F	Find next occurrence of character after Ctrl F	16
Ctrl B	Place cursor at beginning of line	16
Ctrl N	Place cursor at end of line	16
Ctrl L	Change case of character under cursor	16
Ctrl R	Restore line to original form	16
Ctrl C	Exit Edit mode	16
Ctrl Q	Enter line up to cursor position	16
[return]	Enter entire line as it appears	16

LC = Command available on Language Card version only.

SECTION II – OVERVIEW

A. 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, COBAL, 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, off and on. 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" is represented by 00000001 and a "2" is represented by 00000010.

The 6502 microprocessor has five 8-bit registers and one 16-bit register in the ALU (Arithmetic Logic Unit). All data is ultimately handled through these registers. Even this lowest of low-level code requires a program to function correctly. This program is hard wired within the 6502 itself. The microprocessor program functions in three cycles. It fetches an instruction from RAM memory in the computer, decodes it and executes it.

These instructions exist in RAM 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 16 6 front panel switches, each of which, when set on or off, would combine in hex. This requires an additional program in the computer to break the byte down into its respective 8 bits so that 6502 may interpret it.

At the next level up (requiring still more programming), the user may enter his/her data in the form of a three character "mnemonic", a type of code whose characters form an association with the microprocessor operation, e.g., LDA stands for "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 segment 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. And this is what BIG MAC is all about.

* Two 8-Bit registers functioning as a 16-bit register.

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

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

B. BACKGROUND AND FEATURES

BIG MAC 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 the earlier TED II+, in terms of the command mnemonics, pseudo-ops, etc.

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. BIG MAC is no exception. Representing a major step forward, with the addition of macro capability, MAC 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 BIG MAC, in addition to macros, include the use of the logical operators AND, OR, and EOR, and 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 companion program READER 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.

BIG MAC is an editor/assembler which is upwards compatible with TED II + (except for a couple of rarely used opcodes). It has MACRO capability, conditional assembly, and a number of other features missing in TED II+. The names (but not the code) for some of the additional features were borrowed from the Apple assembler and other assemblers to enhance compatibility. This assembler was originally based on a version of TED II +, but has been virtually entirely rewritten. The most obvious differences that a user of TED II+ will notice are the text formatting in edit mode, the speed of the editor (over 20 times faster), and the absence of numerous bugs.

BIG MAC is a coresident assembler, which means that it must contain the source code in memory while assembling. This has several advantages over a disk based assembler, and many people, including the present author, greatly prefer it. However, it has the disadvantage that there is a certain maximum source file size that can be handled in one piece, although this is, in fact, rather large. Every effort has been made to maximize the space available to BIG MAC, and it is suggested that you do likewise by not tying up memory with such utilities as the Program Line Editor when using BIG MAC. Such facilities are very nice in their place, but get in the way of a good assembler or text editor.

BIG MAC assumes that your system has 48K memory and operates on either 3.2 or 3.3 DOS. BEWARE of "custom" DOS's. BIG MAC does an automatic MAXFILES 1 upon entry, then reverts to the usual value on exit.

C. 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 Man.		Synertek 6500-20
Programming the 6502	Rodnay Zaks	Sybex C-202
The Apple Monitors Peeled	Wm. E. Dougherty	Apple Computer, Inc.
A Hex on Thee	Val J. Golding	Peeking at Call—Apple, Vol. II
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.	

ASSEMBLY LINES — *by Roger Wagner*
A continuing series of tutorial articles in SOFTALK magazine. The collected series through Volume 1 will be published separately in the near future. An excellent introduction, easy-to-follow for the beginning assembly language programmer.

CONVERTING BRAND X TO BRAND Y — *by Randall Hyde*
Apple Orchard, Volume 1, No. 1, March/April 80. 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.

SECTION III BIG MAC

A. EXEC MODE

C: CATALOG

After showing the catalog, this command accepts any disk command you wish to give, using standard DOS syntax. Unlike the Load, Append and Save commands, you must type the .S suffix for a source file. This facility is provided primarily for locking and unlocking files. Do not use it to load or save files. If you do not want to give a disk command, just hit return. To cancel a partially typed command use control X or make sure the command is in the wrong syntax (type some commas) or just backspace to the beginning.

L: LOAD

This is used to load a source file from disk. You will be asked for the name of the file. You should not append a .S, since BIG MAC does this automatically. If you have hit L by mistake, just hit <RTN> twice and the command will be cancelled without affecting any file that may be in memory. After a load(or append) command., you are automatically placed in editor mode, just as if you hit E. The source will automatically be loaded to the correct address. Subsequent LOAD or SAVE commands will display the last used file name., 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 (e.g., [RETURN],), the cursor will be placed on the first character of the file name, and you may type in the desired name. [RETURN] alone at this time will cancel the command.

S: SAVE

Use this to save a source file to disk. As in the load command, you do not specify the suffix .S, and you can hit <RTN> to cancel the command. Note that the address and length of the source file are shown on the MENU. These 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 command above, the file name will be displayed and you may type "Y" to SAVE the same file name, or any key for a new file name.

A: APPEND

This loads in a specified source file and places it at the end of the file currently in memory. It operates in the same way as the load 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

When you hit D, the drive used for saving and loading changes from 1 to 2 or 2 to 1. The currently specified drive is shown on the menu. When BIG MAC is first BRUN, the specified drive will be the one used for the BRUN. There is no command to specify slot number, but this can be accomplished by typing C for catalog, then giving the disk command CATALOG, Sn, where n is the slot number.

E: EDITOR

This command places you in the editor/assembler mode. It automatically sets the default tabs for the editor to those appropriate for source files.

Z: ZERO TABS

This is the same as the E command but sets all tabs to zero. This is appropriate for text files that are not source files. This is just a convenience, and the same thing can be accomplished in editor mode by typing TABS <RTN>.

O: SAVE OBJECT CODE

You are permitted to use this command only after the successful assembly of a source file. In this case you will see the address and length of the object code on the menu. As with the source address, this is given for information only. Note that the object address shown is that of the program's ORG (or $8000 by default) and *not* that of the actual current location of the assembled code (which is $8000 or whatever OBJ you have used). When you use 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 it or BRUN it, it will go to that address, which can be anything ($300, $800, etc.) Thus 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 $8000 and above.

Q: QUIT

This exits to BASIC. It also sets up a reentry jump address so that you can enter BIG MAC again by typing CALL 6. This reentry will be a *warm* start, that is, it will not destroy the source file currently in memory. This exit can be used to give disk commands, if that is more convenient than the one provided by C.

B. THE EDITOR

1. Command Mode

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.

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

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

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

HI*mem:* (decimal number between 9472=$2500 and 39584=$9AA0)

This command is rarely needed. It sets the upper limit for the source file and the symbol table produced by the assembler. Its main purpose is to protect the object file area from being used by the symbol table during assembly. (Actually, the symbol table would be destroyed by the object file.) Himem defaults to $8000, and so does not have to be set unless you use a non-default object address. You are not permitted to specify a Himem below $2500, (inside BIG MAC) or above $9AA0.

NEW

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

PR*#* (0-7)

Same function as in BASIC. Mainly used for sending an editor or assembly listing to a printer.

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, because this will likely interfere with BIG MAC's heavy zero page usage.)

TABS number ,number, . . .
TABS number ,number, . . . ,"tab character"

This sets the tabs for the editor, and has no effect on the assembler listing. Up to 9 tabs are possible, but they cannot exceed 39 in value. 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 carriage return will set all tabs to zero.

LEN*gth*

This gives the length in bytes of the source file, and the number of bytes remaining before Himem (usually $8000 — not BASIC Himem).

W*here* (line number)

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

MON*itor*

This exits to the monitor. It sets up return addresses at 0, 6, and at the control Y vector ($3F8). Thus you may reenter by either 6G, 0G or control Y. These reestablish the important zero page pointers from a save area inside BIG MAC itself. Thus control 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. Should you accidentally exit to BASIC with a Ctrl C, don't panic; BIG MAC is very forgiving. Just hit RESET. With the old monitor ROM, you can then type 6G to return to BIG MAC. With the Autostart ROM, RESET will reconnect DOS, and you can then type INT or FP, corresponding to whichever BASIC you are in, then type CALL 6 to return to BIG MAC.

TR*unc*ON

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

TR*unc*OFF

This returns to the default condition of the truncation flag. (This also happens automatically upon entry to the editor from the exec mode or from the assembler.)

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" then you will be presented with the first line in the source which contains a "/" and are placed in edit mode. When you are done editing this line and hit return, assembly will begin. If you use the control C edit abort command, however, you will return to the editor command mode, and any I/O hooks you have established, by PR# etc., will have been disconnected. This will also happen if there is no line with a "/".

Note that by establishing a comment line with "*/" at the beginning, you have a nearly automatic method of keeping track of multiple versions of a program.

D*elete* (line number)
D*elete* (range)
D*elete* (range list)

This deletes the specified lines. Since, unlike BASIC, the line numbers are fictitious, they change with any insertion or deletion. Thus you *must* specify the *higher* range first!

R*eplace* (line number)
R*eplace* (range)

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

L*ist*
L*ist* (line number)
L*ist* (range)
L*ist* (range list)

Lists the source file with added line numbers. Control characters in source are shown in inverse, unless the listing is being sent to a printer or other nonstandard outport. The listing can be aborted by control 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.

P*rint*
P*rint* (line number)
P*rint* (range)
P*rint* (range list)

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

/
/ (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.

Find (d-string)
Find (range) (d-string)
Find (range list) (d-string)

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

Change (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-strings must have the same delimiter with the adjoining ones coalescing. For example, to change occurrences of 'speling" to "spelling" throughout the range 10,100, you would type C20,100"speling" spelling". If no range is specified, the entire source file is used. Before the change operation begins, you are asked whether you want to change "all" or "some". If you select "some" by hitting the "S" key, the editor stops whenever the first string is found and displays the line as it would appear with the change. 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. Control C or key "/" will abort the change process.

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

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

MOVE (range) TO (line number)
MOVE (line number) 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 (line number)
Edit (range)
Edit (range list)
Edit (d-string)
Edit (line number) (d-string)
Edit (range) (d-string)
Edit (range list) (d-string)

This presents the range, etc., line by line to be edited and puts you into edit mode. If a d-string is appended, then only those lines containing the d-string are presented.

For the commands involving a d-string, the character "^" acts as a "WILD CARD". Thus ' "Jon^s" will find both "Jones" and "Jonas".

2. Add/Insert Mode

Add

This places you into the "Add" mode. This acts much like entering 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 control L. This acts as a case toggle, so another control L returns you to upper case mode. (Note that the control L functions differently in EDIT mode; see pg. 16. Also, the exit from "Add" mode is to just hit RETURN as the FIRST character of a 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.) You may also exit the "Add" mode by control X, which then cancels the current line.

Insert (line number)

This allows you to enter text just *above* the specified line. Otherwise, it functions the same as "Add" mode.

Ctrl L

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

Ctrl X

Cancels the current line being entered and returns to EDIT Control Mode.

3. 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 Program Line Editor.

The edit mode commands are:

Control I
: Begins insertion of characters. This is terminated by any control character, such as the arrows or return.

Control D
: Deletes the character under the cursor.

Control F
: Finds the next occurrence of the character typed after the control F.

Control L
: Changes the case of the character under the cursor.

Control O
: Functions as Control I, except inserts any control character (including the command characters such as control O).

Control R
: Returns the line to its original form.

Control Q
: Deletes all characters from the cursor on and completes the edit of that line.

Control C
: Aborts edit mode and returns to the editor's warm start. The current line being edited will retain its original form.

Control B
: Places the cursor at the beginning of the line.

Control N
: Places the cursor one space to the right of the end of the line.

[Return]
: Accepts the line as it appears on the screen and fetches the next line to be edited, or goes to the warm start 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. The reason for this 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, and then editing again and changing the delimiter back to the desired one.

C. THE ASSEMBLER

This 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 to document the features that are available to you in the assembler.

1. Number Format

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

LDA #$100 LDA#$64 LDA #%1100100 LDA #%01100100

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

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

LDA 1000 LDA $3E8 LDA %1111101000

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

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

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

is a good deal more mysterious than its decimal equivalent:

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

2. Source Code Format

A line of source code typically looks like:

LABEL OPCODE OPERAND ;COMMENT

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

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

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

The assembler examines only the first 3 characters of the OPCODE (with the Sweet 16 opcode POPD as the only exception) so, for example, you can use PAGE instead of PAG. (Because of the one exception, the fourth letter should not be a D, however.) The assembler listing will truncate the opcode to 7 letters and will not look well with one longer than 4 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.

3. 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 decimal number
3. A hexadecimal number (preceded by "$")
4. A binary number (preceded by "%")
5. Any ASCII character preceded, or enclosed, by quotes or single quotes
6. The character * (standing for the present address)

All number formats accept 16 bit data and leading zeros are never required. In case 5 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: +, -, /, and *. It also supports the three logical operations:

! = Exclusive OR . = OR & = AND

Thus some examples of legal expressions are:

```
LABEL1–LABEL2
2*LABEL+$231
1234+%10111
"K"–"A"+1
"0" !LABEL
LABEL&$7F
*–2
LABEL.%10000000
```

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

4. Immediate Data

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

#<expression	produces the low byte of the expression
#>expression	produces the high byte of the expression
#expression	also gives the low byte (the 6502 does not accept 2-byte DATA)
#/expression	is optional syntax for the high byte of the expression

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

```
COMPARE   LDX   #FOUND–DATA–1
LOOP      CMP   DATA,X
          BEQ   FOUND
          DEX
          BPL   LOOP
          JMP   REJECT  ;not found
DATA      HEX   E3BC3498
FOUND     RTS
```

With this type of code, if you add or delete some of the "DATA", then the appropriate X-index for the comparison loop is automatically adjusted.

5. Addressing Modes

a. 6502 OPCODES

The assembler accepts, of course, all the 6502 opcodes with standard mnemonics. It also accepts BLT (branch if less than) as an equivalent to BCC, and BGE (branch if greater or equal) as an equivalent to BCS.

There are 12 addressing modes on the 6502. The appropriate syntax for these, in BIG MAC are:

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

Note that there is no difference in syntax for zero page and absolute modes. The assembler automatically uses zero page mode when appropriate. 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 also that the "accumulator mode" does not require (or accept) an operand. 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.

b. SWEET 16 OPCODES

The assembler also accepts all Sweet 16 opcodes with the standard mnemonics. The usual Sweet 16 registers R0 to R15 do not have to be "equated" and the "R" is optional. 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 0D) and not a two byte skip as this would be interpreted by ROM Sweet 16. This is intentional, and is done for internal reasons.

c. PSEUDO OPCODES

i. directives

EQU expression (EQU*als*)
= expression *(optional 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 expression (OR*iG*in)

Establishes the address at which the program is designed to run. It defaults to $8000. Ordinarily there will be only one ORG and it will be at the start of the program. Otherwise the exec mode's "object code save" command will not function.

OBJ expression (OBJ*ect*)

Establishes the address at which the object code will be placed during assembly. It defaults to $8000. There is rarely any need to use this pseudo-op and inexperienced programmers are urged not to use it. You must not set it to an address conflicting with MAC (which includes *all* memory up to the end of the eventual symbol table).

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.

ii. formatting

LST ON or OFF (L*i*ST)

This controls whether the assembly listing is to be sent to the Apple screen and/or other output device. You may, for example, use this to send only a portion of the assembly listing to your printer. Any number of LST instructions may be in the source. If the LST condition is OFF at the end of assembly, then the symbol table will not be printed. The assembler actually only checks the third character of the operand to see whether or not it is a space. Thus, 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, then the object code will be generated much faster, but this is recommended only for debugged programs.

EXP ON or OFF (EXP*and*)

EXP ON will print an entire macro during the assembly. The OFF condition will print only the PMC pseudo-op. EXP defaults to ON.

PAU (PAU*se*)

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 (PAG*e*)

This sends a formfeed ($8C) to the printer. It has no effect on the screen listing.

AST expression (AST*erisks*)

This sends Asterisks to the listing, the same number as the value of the operand. The number format is the usual one, so that AST 10 will send (decimal) 10 asterisks, for example. The number is treated modulo 256 with 0 being 256 asterisks! This differs from TED II+, which recognizes the operand as a hex expression, thus will need to be converted.

SKP expression (SK*i*P)

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

iii. strings

ASC d-string (ASC*ii*)

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 code) the single quote ['] will produce a string with the high bits on, otherwise the high bits will be off. Thus, 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 d-string (*Dextral Character Inverted)*

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

INV d-string (INV*erse*)

This puts a delimited string in memory in inverse format. All choices of delimiter have the same effect.

FLS d-string (FL*aSh*)

This puts a delimited string in memory in flashing format. All choices of delimiter have the same effect.

iv. data and allocation

DA expression (*Define Address)*

This stores the (two byte) value of the operand, usually an address, in the object code, low byte first. DA $FDF0 will generate F0 FD.

DDB Define Double Byte

As above, but places high byte first.

DFB expression (*DeFine Bytes*)

This puts the bytes specified by the operand into the object code. It accepts several bytes of data, which must be separated by commas and contain no spaces. The standard number format is used and arithmetic is done as usual. The "#" symbol is acceptable but ignored, as is "<". The ">" symbol may be used to specify the high byte of a label, otherwise the low byte is always taken. The ">" symbol should appear only as the first character of an expression or immediately after #. That is, the instruction DFB >LAB1−LAB2 will produce the high byte of the value of LAB1−LAB2. For example.

DFB $34,100,LAB1−LAB2,%1011,>LAB1−LAB2

is a properly formatted DFB statement which will generate the object code (hex) 34 64 DE 0B 09, assuming that LAB1=$81A2 and LAB2=$77C4.

HEX 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 each having two hex digits (e.g., 0F, not F). They may be separated by commas or may be adjacent. An error message will be generated if the operand contains an odd number of digits or ends in a comma or, as in all cases, contains more than 64 characters.

DS expression (*Define Storage)*

This does not generate any code, but simply adjusts the object code pointer so that an area equal to the value of the operand is reserved.

v. conditionals

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 nonzero 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 to go on a ROM chip, then you would want one version for the ROM and another, with small differences, to create a RAM version for debugging purposes. 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 8 deep (possibly with some ELSEs between). If some DO condition is off (value 0), then assembly will not resume until its corresponding FIN is encountered, or an ELSE at this level occurs. Nested DO,FIN structures are valuable for putting conditionals in MACROS.

ELSE

This inverts the assembly condition (ON-->OFF OR OFF--> ON) for the last DO.

FIN

This cancels the last DO.

MAC (MAC*ro*)

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. (Thus, things like DA NAME will not be acepted if NAME is the label on a MAC. However, the same thing can be simulated by preceding the macro by a LABEL EQU *, or a LABEL DS 0, etc. There is rarely any need to do this.) See the section on MACROS for details of the usage of macros.

EOM (*End Of Macro)*
<<< *(alternative syntax)*

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

PMC macro name (P*ut Ma*C*ro)*
>>> macro name *(alternative 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.

6. Variables

LABELS beginning with "] " are regarded as VARIABLES. These may be defined only by EQU and cannot be used to label something else. They can be redefined as often as you please. 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.

D. MACROS

1. Defining a macro

A macro definition begins with

NAME MAC (no operand)

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

You can simply define the macro the first time you wish to use it in the program. However, it is preferable to first define all macros at the start of the program with the assembly condition OFF and then refer to them when needed.

Macros cannot be nested. Memory restrictions are too stringent to be able to provide this rarely useful ability. An attempt to nest macros will result in the NESTED MACROS error message.

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.

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

Labels inside macros, such as LOOP and OUT in the example on page 29, are updated each time PMC is encountered. Thus they may be used to branch into the middle of a macro as long as the branch is a "backward" one.

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

Since additional space is available in the language card version of BIG MAC, macro nesting is allowed.

2. Special Variables

Eight variables, named]1 through]8, are predefined and are designed for convenience in MACROS. These are used in a PMC statement as follows: 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   $10
      DO    0
SWAP  MAC
      LDA   ]1
      STA   ]3
      LDA   ]2
      STA   ]1
      LDA   ]3
      STA   ]2
      <<<
      FIN
      >>>   SWAP $6,$7,TEMP
      >>>   SWAP $1000,$6,TEMP
```

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

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

The assembler will accept some other characters in place of the space between the macro name and the expressions in a PMC statement. For example, you may use "/", ",", "—", or "(". The commas are required, however, and no extra spaces are allowed.

3. Sample Program

On the next page is a sample program intended to illustrate the usage of macros with nonstandard variable. It would, however, be simpler and more pleasing if it used]1 instead of]MSG. (In that case the variable equates should be eliminated and the values for]1 must be specified in the >>> lines.)

```
HOME            EQU   $FC58
COUT            EQU   $FDED
KEY             EQU   $C000
STROBE          EQU   $C010
DOS             EQU   $3D3
                DO    0           Assembly off
SENDMSG         MAC               ;Start of definition of the macro "SENDMSG"
                LDY   #0
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
IMSG            EQU   HITMSG
                ]]]   SENDMSG
GETKEY          LDA   KEY         Get input
                BPL   GETKEY      Key hit yet?
                BIT   STROBE      Yes, set up for next one
                CMP   #"F"
                BNE   INVRS
IMSG            EQU   FMSG        If so, then it will do this
                ]]]   SENDMSG
INVRS           CMP   #"I"
                BNE   NORM
IMSG            EQU   IMSG
                ]]]   SENDMSG
NORM            CMP   #"N"
                BNE   STP
IMSG            EQU   NMSG
                ]]]   SENDMSG
STP             CMP   #"S"        Does he want to stop?
                BNE   GETKEY      No, get the next input
                JMP   DOS         All done, exit gracefully
HITMSG          ASC   !HIT        A KEY: "F","I","N", OR "S"!
                HEX   8D8D00
FMSG            FLS   "THIS IS A FLASHING MESSAGE"
                HEX   8D8D00
IMSG            INV   "THIS IS A MESSAGE IN INVERSE"
                HEX   8D8D00
NMSG            ASC   "THIS IS A NORMAL MESSAGE"
                HEX   8D8D00
```

E. SYMBOL TABLE

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

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

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

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

Also, bit 6 is set during the alphabetical printout to flag printed symbols, then removed during the numerical order printout. The symbol printout is formatted for an 80 column printer, or for one which will send a carriage return after 40 columns.

F. TECHNICAL INFORMATION

1. Important Addresses

BIG MAC's address and length are A$803,L$1CFB. You should never BSAVE BIG MAC with a source file in memory, use NEW in editor mode first. SOURCE is placed at $2500 when loaded, regardless of its original address.

The important pointers, as in TED II+ are:

START OF SOURCE	in	$A,$B	(always set to $2500)
HIMEM	in	$C,$D	(defaults to $8000)
END OF SOURCE	in	$E,$F	

When you exit to BASIC or to the monitor, these pointers are saved at $809–$80E. They are restored upon reentry to BIG MAC.

Warm entry to EXEC Mode is $806=2054. This is the entry given by a CALL 6, if you used "Q" to exit from EXEC mode to BASIC. Entry via $803=2051 will delete the default file name, but is otherwise the same as a warm start entry. Warm entry directly to the editor is available by typing C18G, but its use is not recommended as it is possible to lose certain pointers.

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

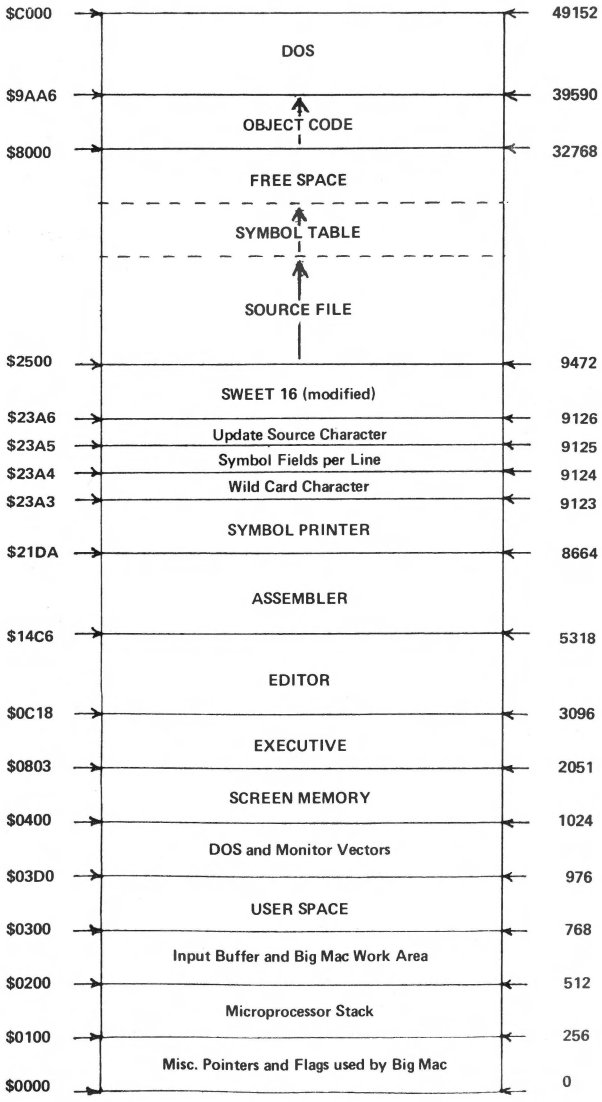
There are three bytes of data you may wish to change to suit your own needs. For that reason, these have been placed in convenient locations, just below Sweet 16 as shown on the memory map.

$23A5 (SWEET–1) Holds the character "/" ($AF), which is searched for by the "Update Source?" entry to the assembler module.

$23A4 (SWEET–2) Holds the number (currently 4) of columns that are printed in the symbol table before a carriage return is sent. This can be changed to accommodate printers with greater or lesser line lengths. Each symbol column represents 20 printer columns.

$23A5 (SWEET–3) Holds the "Wild Card" character "^" ($DE), used by the editor in F(ind) and other search routines.

2. Memory Map



3. In Case of a Crash

Sometimes, because of static electricity spark or other cause, BIG MAC, or any program, may crash. Because of this possibility, you should save your programs at regular intervals while writing them. In case of a crash, however, you may be able to revive your source by the following procedure.

Exit from BIG MAC (hit RESET if necessary).
Go to the monitor by CALL -151.
Write down the two bytes at $80D,$80E. (If you are operating from Integer Basic, which is generally preferable, then use the bytes at $E,$F instead!)
BLOAD BIG MAC (just in case), *DO NOT BRUN BIG MAC.*
Replace the two bytes you wrote down at $80D,$80E. (Not $E,$F.)
Type 803G (RTN) to go to BIG MAC's warm start.
Type E to get to the editor.
Type W0, which will print out the end of source in hex.
If this number looks reasonable, then list the source and check it out.
If not, then type MON to go to the monitor. Try to find the end of source in memory (one past the last $8D). If assembly had been started, then there should be an $FF at the end of source. Set $80D,$80E to the address of the end of source (low byte in $80D). Type control Y to return to BIG MAC. If things are still not OK , then you have problems.

If DOS has been clobbered, then you have further problems. In this case, instead of doing the above things, write down the bytes at $80D,$80E (or $E,$F), reboot from a slave disk *not* a master disk), BRUN BIG MAC, go to the monitor and reestablish $80D,$80E as above, and from your data, and finally type control Y to return to BIG MAC and check things out. For this reason, BIG MAC is supplied as a 48K *slave* diskette.

4. Using TED II+ Source Files With BIG MAC

You may simply load in a file created by TED II+ and, in most cases, assemble it with BIG MAC. Some minor changes to opcodes may be necessary. In particular, you should usually remove any OBJ opcodes, as these are rarely of use in BIG MAC.

Also, the editor in BIG MAC has some refinements that will not be used by this procedure. The file can be converted to the form it would have if written by BIG MAC simply by running through the entire source in edit mode, hitting return at each line. For a large source however, a special program, FIX has been provided that does this automatically. Simply load in your file, to to EXEC mode, hit C for the catalog, and BRUN FIX. The fix will be done almost instantly, and will return automatically to BIG MAC.

Among other things, FIX removes extra spaces at the ends of lines, which sometimes results in substantial savings in memory. It does not remove extra spaces within lines, so that it can be used with text files which are not source files. However, this can be done by typing, from the editor, C" " " several times until no further changes are made. You should be somewhat careful with this since you do not want to make such changes in comments and ASCII strings.

G. ERROR MESSAGES

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

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

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

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

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

MEMORY FULL
On the first pass, the symbol table exceeds HIMEM ($8000 by default). Assembly is aborted by this error.

UNKNOWN LABEL
Your program refers to a LABEL that does not exist. This also occurs if you try to reference a MACRO definition by anything other than PMC. This 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 to a label that is not a MACRO.

NESTED MACROS
A MACRO definition inside another one, or a PMC inside a MACRO definition.

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

H. LANGUAGE CARD VERSION

If you have the Language Card version of BIG MAC, all differences between it and the standard version are detailed in the Language Card Supplement, which is bound in, or included with this manual. This section should be read at once, as a number of additional functions have been included, and certain commands have been modified.

SECTION IV OTHER PROGRAMS

A. SOURCEROR

1. Introduction

Sourceror is a sophisticated and easy to use disassembler designed as a subsidiary to the BIG MAC assembler. It will make BIG MAC source files out of binary programs, usually in a matter of minutes. Sourceror disassembles Sweet 16 code as well as 6502 code.

The main part of Sourceror is called SRCRR.OBJ, but this cannot be run (conveniently) directly, since it may overwrite DOS buffers and bomb the system. For this reason, a small program named SOURCEROR is provided. This runs in the input buffer, thus 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 $8A00-$9AA5). To mimimize the possibility of accident, SRCRR.OBJ has a default location of $4000 and if you BRUN it, it will just return without doing anything. If you try to BRUN it at its designed location of $8A00, however, you could be in for big trouble. Sourceror assumes the standard Apple screen is being used and will not function with an 80 column card.

2. 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 $800 or $803. Small programs at $4000 and above, or medium sized ones above $6000 will probably be OK at their original locations.

2. BRUN SOURCEROR

3. You will be told that the default address for the source file is $2500. This was selected because it is the address used by BIG MAC (this is not important, however) and 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 Control 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. Otherwise, you must specify, in hex, the *present* location of the code to be disassembled. Then, you will be asked to give the *ORIGINAL* location of that program.

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

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

At this point, and until the final processing, you may hit RESET to return to the start of the Sourceror program. If you then hit RESET once more, you will exit Sourceror and return to BASIC. (This assumes you are using the autostart monitor.)

3. 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 wish to 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.

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 Sweet 16 mode.

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

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

S (Sweet)

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

N (Normal)

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

H (Hex)

This creates the HEX data opcode. It defaults to one byte of data. If you insert a one byte (one or two digits) hex number after the H, then 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 charcters have been treated, or when the high bit of the data changes. In the latter case 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 (1 or 2 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 (while in Sourceror, not in BIG MAC) as flashing letters.

W (Word)

This disassembles the next two bytes at the current location as a DA opcode. Optionally, if the command WW is used, then these bytes are disassembled as a DDB opcode. Finally, if W- is used as the command, the two bytes are disassembled in the form DA LABEL—1. (The latter is often the appropriate form when the program uses the address by pushing it on the stack. You may detect this while disassembling, or only 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.)

4. Housekeeping Commands

/ (Cancel)

This essentially cancels the last command. More exactly, it reestablishes 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 back up is desired. For example: suppose you type T to disassemble some text. You may not know what to expect the following the text, so you can just type L to look at it. Then if, for example, the text turns out to be followed by some hex data (such as $8D for a carriage return), then 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 $1234 to $1333 type 1234R. 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. Thus 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.

I (Instructions)

This prints the title page out to remind you of the available commands.

Q (Quit)

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

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

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

5. 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 you don't, it will be lost.) 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. Thus, replace the disk first if you want the source to go on another disk.

To look at the disassembled source, BRUN BIG MAC and load it in.

6. 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, this 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 or JMP's or JSR's. This is almost a sure sign of an error (yours, not Sourceror's) in the disassembly. It may have resulted from an attempt to disassemble a data area as regular code. Note that 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, then you will not get an error, but the assembly MAY NOT BE CORRECT. Thus, it is important to deal with this situation first. The trouble here is for the following reason. If, for example, the disassembler finds the data AD 00 8D, it will disassemble it, correctly, as LDA $008D. The assembler, however always assembles this code as a zero page instruction, giving the two bytes AD 8D. Occasionally you will find a program that uses this form for a zero page instruction. In that case, you will have to change it to HEX data to have it assemble identically to its original form. More usually, however, it was data in the first place, rather than code, and must be dealt with to get a correct assembly.

7. The Memory Full Message

When the source file reaches within $600 of the start of Sourceror (that is, when it goes beyond $8400) you will see "MEMORY FULL" and "HIT A KEY" in flashing format. When you hit a key, Sourceror will go directly to the final processing. The reason for the $600 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 casue problems since the front end of Sourceror will not be used again at that point. There is a "secret" override provision at the memory full point. If the key you hit is Control 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 farther 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 reestablished upon exit, and have partially destroyed sourceror.

B. HELLO AND GREETINGS

Initially, this diskette has two programs named "Hello". On booting or running Hello the first time, a fancy title program will be seen. When the program has been executed, it will delete itself, and a subsequent boot or RUN HELLO will run the second hello program, which in turn will BRUN BIG MAC. The original Hello Program also appears on the diskette under the title "Greetings".

C. THE TEXT FILE COMPANIONS

1. Introduction

A pair of programs, named READER and WRITER, respectively, offer you the opportunity to use BIG MAC as a limited text editor, and in addition, permit you to read in source files created in text file format by other assemblers, such as Apple Computer's DOS Tool Kit, then convert to proper BIG MAC syntax.

2. READER

This program will read any sequential text file into BIC MAC's edit buffer, where the BIG MAC editor may be used for editing, just as you would a regular source file. When the text has been edited, it may be saved as a binary file, using MAC's normal S(ave) command, or by using WRITER, it may be written back to disk as a text file.

Any sequential text file may be read in and edited, for example: EXEC, data, assembler source, etc. READER will add a "T" prefix to the file name you supply, thus if your text file does not already have this prefix, it must be renamed prior to calling READER. Later, you may rename it to its original name, if desired. Alternate drives or slots may be specified by appending "D?" or appropriate parameter to the file name given to READER. Lines longer than 256 characters ($100 bytes) will be split into two lines, and should be taken into account.

INSTRUCTIONS:

1. BRUN BIG MAC
2. Hit "C" for catalog
3. After the "COMMAND:" prompt, type:
   BRUN READER
4. READER will then ask you for the file name to be loaded, which should then be entered.
5. When the load has been completed, you will be returned to MAC's EXEC mode. Type "E" to enter the editor.

3. WRITER

This program writes a BIG MAC file into a text file. It makes no difference what the original form of the file was, once it is in MAC's edit buffer, WRITER will output it as a text file.

Before using Writer, make certain the file is intact, by using the editor's L(ist) or P(rint) commands, since WRITER will delete the original file if you supply WRITER with the same name. Just like READER, WRITER appends a "T" prefix to the file name, and similarly, you may specify alternate drive or slot parameters by adding the proper S or D values to the file name.

INSTRUCTIONS

1. BRUN BIG MAC (if required)
2. L(oad) or create a file
3. Hit "C" for catalog
4. After the "COMMAND:" prompt, type:
   BRUN WRITER
5. WRITER will then ask you for the name of the file to be saved, which should then be entered.

D. THE MACRO LIBRARY

A macro library with three example macro programs is included in source file form on this diskette. The purpose of the library is to provide some guidance to the newcomer to macros and how they can be used within an assembly program. Note all macros are defined at the beginning of the source file, then each example program places the macros where they are needed. Conditionals are used to determine which example program is to be assembled.

E. AUTOSTART ROM SUPPLEMENT

The "Supplement" is a group of utility routines by Steve Wozniak that reside in the F4 and F8 ROMs of a standard Apple] [. They are not present in the Apple] [Plus, nor in an Apple] [that has had the "old" Monitor ROM replaced with the Autostart ROM. Instructions for using Single Step, Trace, and the Mini-Assembler can be found in the Apple] [reference manual. Wozniak's programs have been modified by Guil Banks who, in addition, wrote the included "Convert" program. Complete documentation for the Supplement is also contained in the program SUPPLEMENT.DOC.

The Supplement may be entered from Monitor via the Ctrl Y user function, and requires that your BASIC hello program contain the two following lines:

```
100   HIMEM:-29440:REM $8D00
110   POKE 1016,76:POKE 1017,0:POKE
      1018,141:REM SET CTRL Y JUMP
      AT 3F8 TO $8D00
```

The Supplement's prompt character is a ")", and all Monitor commands may be entered from within the supplement, except that the Mini-Assembler is enabled by entering a Ctrl A. A slash "/" exits the Mini-Assembler and returns you to the Supplement. The Trace function is also terminated with the slash. In Trace mode, a Ctrl S will cause the listing to pause until another key is hit.

The "Convert" routine will convert a hexadecimal number in the range 0-FFFF to decimal, and display it in both normal and two's complement decimal equivalent form, and will convert a decimal number in the range 0--65535 to hexadecimal and hexadecimal two's complement, displaying each. Convert is entered with a Ctrl T, and exited with the slash. Convert mode is identified by a carat "^" prompt character.

To enter decimal to hex mode, type "^T" followed by a carriage return.

To enter hex to decimal mode, type "^H" followed by a carriage return.

Convert will remain in the conversion mode set by the T or H parameters until the alternate parameter is entered, or until a slash is typed. Note that control returns to the Monitor when any function other than those in the Supplement is used. Convert does not require or allow negative number inputs.

CONVERSION EXAMPLES

^T [c/rtn] ^936 [c/rtn] displays: 03A8 FC58
^H [c/rtn] ^FC58 [c/rtn] displays: 64400 936

F. SWEET 16

1. 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 a low level language, with a strong likeness to conventional 6502 Assembly language.

To use Sweet 16, you must learn the language – and to quote "WOZ", "The op code list is short and uncomplicated". "WOZ", 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 op codes and instruction sets, and uses the SAVE and RESTORE routines from the Apple Monitor to preserve the 6502 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 6502 Dream Machine", first appeared in Byte Magazine, November 1977 and later in the original "WOZ PAK". This article is included here 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.

Finally, the friendly help of A.P.P.L.E. is available to hand-hold and counsel your adventures with Sweet 16.

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" ($300) at line 120, enters a 6502 machine language subroutine having the single purpose of entering a Sweet 16 subroutine and subsequently returning to BASIC (addresses $300,$301,$302, and $312 respectively). The Sweet 16 subroutine of course performs the move, and is entered at hex locations $303 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 at $800, which is the fixed source of data to be moved (in this case, the string A$).

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 $0000 (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 the purposes of the convenience 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, the high order byte only is entered, and the program allows a choice between Decimal 9 and the H.O.B. of program pointer 1, to avoid unnecessary problems (in this demonstration enter a decimal number between dec 9 and 144 for a 48K APPLE).

The 8 bytes of data displayed starting at $00 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 6502 Dream Machine", it will be remembered that Sweet 16 can establish 16 double byte registers, starting at $00. That 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 $0000 is large enough for this purpose. These four registers are established as R0, R1, R2, and R3:

R0	$0000	&	0001	— Sweet 16 accumulator
R1	$0002	&	0003	— Source address
R2	$0004	&	0005	— Destination address
R3	$0006	&	0007	— Number of bytes to move
.				
.				
.				
R14	$001C	&	001D	— Prior result register
R15	$001E	&	001F	— Sweet 16 program counter

Additionally, an examination of registers R14 and R15 will extend an understanding of Sweet 16, being fully explained in the "WOZ" text. Notice that the HOB of R14, (located at $1D) contains $06, which is the doubled register specification(3X2 =$06).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 $0312 when execution ended, and 6502 machine code resumes.

To try a sample run, enter the Integer basic program as shown in Listing #1. Of course, REM statements can be omitted, and line 10 is only helpful if the machine code is to be stored on disk.

The Listing #2 must also be entered starting at $300.

Note that a 6502 disassembly does not look like Listing #3, but the included Sourcerer disassembler would create a correct dissassembly.

Enter RUN and RETURN

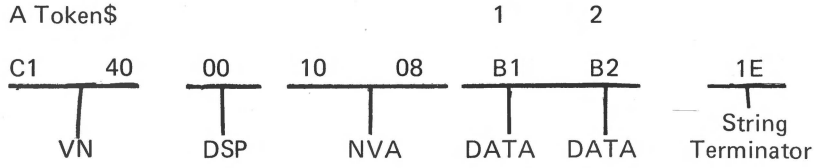
Enter 12 and hit RETURN (A$ – A$ string data)
Enter 8 and hit RETURN (High order byte of destination)

The display should appear as follows:

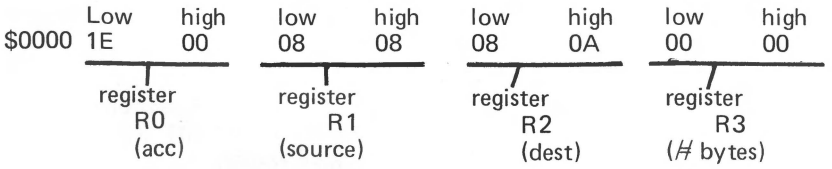
```
$0800–C1 40 00 10 08 B1 B2 1E (SOURCE)
$0A00–C1 40 00 10 08 B1 B2 1E (DEST.)
$0000–1E 00 08 08 08 0A 00 00 (Sweet 16)
```

Note that the 8 bytes stored at $0A00 are identical to the 8 bytes starting at $0800, indicating an accurate move of 8 bytes length has been made. It will be seen that they are moved one byte at a time starting with token C1 and ending with token 1E (if moving less than 8 bytes, the data following the moved data would be whatever exists at those locations before the move).

The bytes have the following significance:



The Sweet 16 registers are shown:



The low order byte of R0, the Sweet 16 accumulator, has $1E in it, which was 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 LOB of R2 was incremented exactly like R1.

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

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

Working with this demonstration program, and study of the text material will soon enable the writing of Sweet 16 programs to perform additional 16 bit manipulations. The unassigned op codes 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.

For "Appleites" who desire to learn machine language programming, Sweet 16 can be used as a starting point. Having less op codes to learn and excellent text material to use, one could be effective very soon.

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

LISTING #1

```
]LIST
   10  PRINT "[D]BLOAD SWEET": REM CTRL
       D
   20  CALL -936: DIM A$(10)
   30  INPUT "ENTER STRING A$",A$
   40  INPUT "ENTER # BYTES ",B
   50  IF NOT B THEN 40: REM AT LEAST 1

   60  POKE 778,B: REM POKE LENGTH
   70  INPUT "ENTER DESTINATION",A
   80  IF A) PEEK (203)-1 THEN 70
   90  IF A< PEEK (205)+1 THEN 70
  100  POKE 776,A: REM POKE DESTINATION

  110  M=8: GOSUB 160: REM DISPLAY
  120  CALL 768: REM GOTO $0300
  130  M=A: GOSUB 160: REM DISPLAY
  140  M=0: GOSUB 160: REM DISPLAY
  150  PRINT : PRINT : GOTO 30
  160  POKE 60,0: POKE 61,M
  170  CALL -605: RETURN : REM XAM8 IN
       MONITOR
```

LISTING #2

Enter code as follows:

```
                                          A     B
300:20 89  F6  11 00 08 12 00 00 13 00 00 41 51 52
   F3 07 FB 00 60
```

LISTING #3

Sweet 16

```
$300  20  89  F6  JSR   $F689
$303  11  00  08  SET   R1  Source address
$306  12  00  00  SET   R2  destination address
                  A
$309  13  00  00  SET   R3  length
          B
$30C  41      LD    @ R1
$30D  52      ST    @ R2
$30E  F3      DCR   R3
$30F  07  FB  BNZ   $30C
$311  00      RTN
$312  60      RTS
```

Data will be poked from the Integer Basic program –

"A" from Line 100
"B" from Line 60

2. SWEET 16: A Pseudo 16 Bit Microprocessor

By Steve Wozniak

a. DESCRIPTION

While writing APPLE BASIC for a 6502 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 6502 would have only slightly alleviated the problem. What I really needed was a 6502/RCA 1800 hybrid — a powerful 8-bit data handler complemented by an easy to use processor with 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 (R0-R15), actually 32 memory locations. R0 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 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 1 byte of code each. The non-register ops are primarily 6502 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 SWEET 16 PC, effecting a branch.

SWEET 16 is intended as a 6502 enhancement package, not a stand-alone processor. A 6502 program switches to SWEET 16 mode with a subroutine call and subsequent code is interpreted as SWEET 16 instructions. The non-register op RTN returns the user program to 6502 mode after restoring the internal register contents (A, S, Y, P, and S). The following example illustrates how to use SWEET 16.

```
300   B9  00  02          LDA   IN,Y      get a char.
303   C9  CD              CMP   #"M"      "M" for move
305   D0  09              BNE   NOMOVE    No. skip move
307   20  89  F6          JSR   SW16      Yes, call SWEET 16
30A   41          MLOOP   LD    @R1       R1 holds source addr.
30B   52                  ST    @R2       R2 holds dest. addr.
30C   F3                  DCR   R3        Decr. length
30D   07  FB              BNZ   MLOOP     Loop until done
30F   00                  RTN             Return to 6502 mode.
310   C9  C5    NOMOVE    CMP   #"E"      "E" char?
312   D0  13              BEQ   EXIT      Yes, exit
314   C8                  INY             No, cont.
```

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

b. 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. Thus LD and ST are opcodes 2N and 3N respectively, while LD @ and ST @ are codes 4N and 5N.

Opcodes 0 to C (hex) are assigned to the thirteen non-register ops. Except for RTN (opcode), BK (0A), and RS (0B), the non-register ops are 6502 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.

c. 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

00	RTN		(Return to 6502 mode)
01	BR	ea	(Branch always)
02	BNC	ea	(Branch if No Carry)
03	BC	ea	(Branch if Carry)
04	BP	ea	(Branch if Plus)
05	BM	ea	(Branch if Minus)
06	BZ	ea	(Branch if Zero)
07	BNZ	ea	(Branch if NonZero)
08	BM1	ea	(Branch if Minus 1)
09	BNM1	ea	(Branch if Not Minus 1)
0A	BK		(Break)
0B	RS		(Return from Subroutine)
0C	BS	ea	(Branch to Subroutine)
0D			(Unassigned)
0E			(Unassigned)
0F			(Unassigned)

d. REGISTER INSTRUCTIONS

SET Rn, Constant [1n] **Low** **High** **(Set)**

The 2-byte constant is loaded into Rn (n=0 to F, hex) and branch conditions set accordingly. The carry is cleared.

EXAMPLE

```
15 34 A0      A0   SET   R5,$A034    R5 now contains $A034
```

LD Rn [2n] **(Load)**

The ACC (R0) 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 A0      A0   SET   R5,$A034
25                 LD    R5          ACC now contains $A034
```

ST Rn [3n] **(Store)**

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
```

LD @Rn **4N** **(Load Indirect)**

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

```
                                  Acc is loaded from mem
15 34 A0    SET   R5,$A034        location $A034
45          LD    @R5             R5 is incr to $A035
```

ST @Rn **5n** **(Store indirect)**

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 A0    SET   R5,$A034        Load pointers R5, R6 with
16 22 90    SET   R6,$9022        $A034 and $9022
45          LD    @R5             Move byte fr $A034 to $9022
56          ST    @R6             Both ptrs are incremented
```

LDD @Rn **6N** **(Load double-byte indirect)**

The low order ACC byte is loaded from the 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

```
                                  The low-order acc byte is
                                  loaded from $A034,
15 34 A0    SET   R5,$A034        high-order from $A035
65          LDD   @R5             R5 is incremented to $A036
```

STD @Rn **7n** **(Store double-byte indirect)**

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 A0  SET   R5,$A034    Load pointers R5, R6
16 22 90  SET   R6,$9022    with $A034 AND $9022
65        LDD   @R5         Move double byte from
76        STD   @R6         $A034-35 to $9022-23.  Both
                            pointers incremented by 2
```

POP @Rn **8n** **(Pop indirect)**

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 1. 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 A0   SET   R5,$A034    Init stack pointer
10 04 00   SET   R0,4        Load 4 into acc
55         ST    @R5         Push 4 onto stack
10 05 00   SET   R0,5        Load 5 into acc
55         ST    @R5         Push 5 onto stack
10 06 00   SET   R0,6        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
```

STP @Rn [9n] **(STORE POP indirect)**

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 A0    SET   R4,$A034    Init pointers
15 22 90    SET   R5,$9022
84          POP   @R4         Move byte from
95          STP   @R5         $A033 to $9021
84          POP   @R4         Move byte from
95          STP   @R5         $A032 to $9020
```

ADD Rn [An] **(Add)**

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

EXAMPLE

```
10 34 76    SET   R0,$7634    Init R0 (acc) and R1
11 27 42    SET   R1,$4227
A1          ADD   R1          Add R1 (sum=B85B, c clear)
A0          ADD   R0          Double acc (R0) to $70B6
                              with carry set
```

SUB Rn **Bn** **(Subtract)**

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

$$ACC = ACC + \overline{Rn} + 1$$

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

```
10 34 76    SET  R0,$7634    Init R0 (acc)
11 27 42    SET  R1,$4227    and R1
B1          SUB  R1          Subtract R1
                             (diff=$340D with c set)
B0          SUB  R0          Clears acc. (R0)
```

POPD @Rn **Cn** **(Pop Double-byte indirect)**

Rn is decremented by 1 and the high-order ACC byte is loaded from the memory 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 A0    SET  R5,$A034    Init stack pointer
10 12 AA    SET  R0,$AA12    Load $AA12 into acc.
75          STD  @R5         Push $AA12 onto stack
10 34 BB    SET  R0,$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 stack
```

CPR Rn **Dn** **(Compare)**

The ACC (R0) 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 A0            SET  R5,$A034   Pointer to mem
16 BF A0            SET  R6,$A0BF   Limit address
B0          LOOP1   SUB  R0         Zero data
75                  STD  @R5        Clear 2 locns
                                    Increment R5 by 2
25                  LD   R5         Compare pointer R5
D6                  CPR  R6         to limit R6
02 FA               BNC  LOOP1      Loop if c clear
```

INR Rn **En** **(Increment)**

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

EXAMPLE

```
15 34 A0      SET  R5,$A034   (Pointer)
B0            SUB  R0         Zero to R0
55            ST   @R5        Clr Locn $A034
E5            INR  R5         Incr R5 to $A036
55            ST   @R5        Clrs locn $A036
                              (not $A035)
```

DCR Rn **Fn** **(Decrement)**

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 A034)

```
15 34 A0            SET  R5,$A034   Init pointer
14 09 00            SET  R4,9       Init counter
B0                  SUB  R0         Zero acc.
55          LOOP2   ST   @R5        Clear a mem byte
F4                  DCR  R4         Decrement count
07 FC               BNZ  LOOP2      Loop until zero
```

e. NON-REGISTER INSTRUCTIONS

RTN [00] **(Return to 6502 mode)**

Control is returned to the 6502 and program execution continues at the location immediately following the RTN instruction. The 6502 registers and status conditions are restored to their original contents (prior to entering Sweet 16 mode).

BR ea [01] [d] **(Branch always)**

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 twos complement value from -128 to +127. Branch conditions are not changed.

Note that the effective address calculation is identical to that for 6502 relative branches. The hex add & subtract features of the APPLE] [monitor may be used to calculate displacements.

d	=	**$80**	**ea**	=	**PC**	+	**2**	- **128**
d	=	**$81**	**ea**	=	**PC**	+	**2**	- **127**
d	=	**$FF**	**ea**	=	**PC**	+	**2**	- **1**
d	=	**$00**	**ea**	=	**PC**	+	**2**	+ **0**
d	=	**$01**	**ea**	=	**PC**	+	**2**	+ **1**
d	=	**$7E**	**ea**	=	**PC**	+	**2**	+ **126**
d	=	**$7F**	**ea**	=	**PC**	+	**2**	+ **127**

EXAMPLE

```
$300:  01  50      BR $352
```

BNC ea [02] [d] **(Branch if No Carry)**

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.

BC ea [03] [d] **(Branch if Carry set)**

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

BP ea [04] [d] **(Branch if Plus)**

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 A034 to A03F)

```
15  34  A0              SET   R5,$A034    Init pointer
14  3F  A0              SET   R4,$A03F    Init limit
B0          LOOP3       SUB   R0
55                      ST    @R5         Clear mem byte
            *                             Increment R5
24                      LD    R4          Compare limit
D5                      CPR   R5          to pointer
04  FA                  BP    LOOP3       Loop until done
```

BM ea [05] [d] **(Branch if Minus)**

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

BZ ea [06] [d] **(Branch if Zero)**

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

BNZ ea [07] [d] **(Branch if NonZero)**

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

BM1 ea [08] [d] **(Branch if Minus 1)**

A branch is effected only if the prior 'result' was minus 1 ($FFFF hex). Branch conditions are not changed.

BRK [0A] **(break)**

A 6502 BRK (break) instruction is executed. Sweet 16 may be reentered non destructively at SW16d after correcting the stack pointer to its value prior to executing the BRK.

RS [0B] **(Return from Sweet 16 subroutine)**

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.

BS ea [OC] [d] **(Branch to Sweet 16 subroutine)**

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 A034-A03B to 3000-3007)

```
15  34  A0          SET   R5,$A034    Init pointer 1
14  3B  A0          SET   R4,$A03B    Init limit 1
16  00  30          SET   R6,$3000    Init pointer 2
0C  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
04  FA              BP    MOVE
0B                  RS                ;Return
```

f. THEORY OF OPERATION

Sweet 16 execution mode begins with a subroutine call to SW16. The user must insure that the 6502 is in hex mode upon entry. All 6502 registers are saved at this time, to be restored when a Sweet 16 RTN instruction returns control to the 6502. If you can tolerate indefinite 6502 register contents upon exit, approximately 30 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 6502 registers, Sweet 16 initializes its PC (R15) with the subroutine return address off the 6502 stack. Sweet 16's PC prints 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 6502 instructions. The main loop at SW16B repeatedly calls the 'execute instruction' routine at SW16C which examines on opcode for type and branches to the appropriate subroutine 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 0 OP with OP between 0 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 R14 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.

g. 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 6502 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
```

To save code, the subroutine entry address (minus 1) is pushed onto the stack, high-order byte first. A 6502 RTS (Return from Subroutine) is used to pop the address off the stack and into the 6502 PC (after incrementing by 1). The net result is that the desired subroutine is reached by executing a subroutine return instruction!

h. OPCODE SUBROUTINES

The register op routines make use of the 6502 '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 R0 (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 thus 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 6502 convention, the low-order data byte precedes the high-order byte.

Most Sweet 16 nonregister 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 6502 register contents, pops the subroutine return stack and jumps indirect through the Sweet 16 PC. This transfers control to the 6502 at the instruction immediately following the RTN instruction.

The BK op actually executes a 6502 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.

i. 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 6502 register contents, and a few levels of the 6502 subroutine return address stack. If you don't need to preserve the 6502 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.

j. USER MODIFICATIONS

You may wish to add some of your own instructions to this implementation of Sweet 16. If you use the unassigned opcodes $0E and $0F, 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 (R0) with the address you wish to jump to (minus 1) and executing a ST R15 instruction.

G.UTILITY 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.

1. The Mini-Assembler

This is a simple, but handy assembler., which does not support labels or line insertion. Each instruction and operand is assembled at entry time, when it is also checked for correct syntax. Its main purpose is for the speedy entry of short programs. The prompt is the exclamation point "!", and instructions on its use may be found on page 66 of the Apple] [reference manual.

2. 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 WOZPAK. (See "Suggested Reading" in Section II.)

3. 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 Apple] [Monitor Peeled", and a multiply demo by Dave Garson is included on this diskette.

4. 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 0-65535. It is easily integrated into 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.

5. MSGOUT

This is a subroutine by Andy Hertzfeld to output ASCII strings from an Assembly Language program. If BIG MAC's INV or FLS Pseudo-ops are used in connection with it, the ORA#$80 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.

6. UPCON

This utility by Glen Bredon is provided for users who do not have a lower case 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 BIG MAC, then BRUN UPCON. This program is integrated in the Language Card version of BIG MAC.

7. FIX

Another utility by Glen Bredon, this removes excess spaces from source files in memory. Some additional information may be found in Section III, F,4, "Using TED] [+ Source Files.

H. 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" isntruction 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 driver so that inverse and flashing characters would not upset the printer when doing a catalog, etc.

Concurrently, many new interface cards of all kinds were developed for the Apple, clock cards, 80 column cards, ROM cards, etc., until card space is now at a premium. Running a serial printer from the game I/O port is one such 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 1, it can be output with or without video, and the video may be left on, even when printing beyond 40 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 BIG MAC, and called with the BIG MAC 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.

SECTION V USING BIG MAC

By T. Petersen

Notes and demonstrations for the beginning BIG MAC programmer

A. INTRODUCTION

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

Many of the BIG MAC 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 base for further independent study on the part of the programmer.

A note of clarification:

Throughout the BIG MAC manual, various uses are made of the terms "mode" and "Module".

In this section, "module" refers to a distinct computer program component of the BIG MAC 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, or (2) The Editor, indicated by the ':' prompt.

EXECUTIVE CONTROL MODE	EDITOR CONTROL MODE
Executive Module	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.

B. 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 BIG MAC, 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.

From the very beginning:

1. BRUN BIG MAC
2. When the '%' prompt appears at the bottom of the EXEC mode menu, type 'E' or 'Z'. This instantly places the system in EDITOR control mode. If entered with a 'Z', then tab stops are set to zero.
3. As we are entering an entirely new program, use the following entry command after the ':' prompt — type 'A' (for ADD) and press return. A '1' appears one line down and to the right, 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 return 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 a '*'. 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' and hit return.
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 '$300', and hit return. Note in most cases the 'OBJ' pseudo-op is neither required nor desirable.
7. On line 3, perform the same sequence: space, type 'ORG', space, type '$300', return.
8. On line 4, do not space once after the line number. Type 'BELL', space, 'EQU', space, '$FBDD', return.
9. Line 5 — Type 'START', space, 'JSR', space, 'BELL', space, ';' (semicolon), 'RING THE BELL', return.

   Semicolons are a convention sometimes used within command lines to mark the start of comments. Unlike the asterisk, they are not required and may be omitted entirely, unless no operand is present.
10. Line 6 — 'END', space, 'RTS', return.
11. The program has been completely entered, but the system is still in ADD mode. To exit ADD just hit return, or type Ctrl X, 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:

```
A
    1  *DEMO PROGRAM 1
    2   OBJ $300
    3   ORG $300
    4  BELL EQU $FBDD
    5  START JSR BELL ;RING THE BELL
    6  END RTS
    7  \
```

13. Now, type 'L', the LIST command, after the '.' prompt, and return.

```
:L

    1  *DEMO PROGRAM 1
    2             OBJ   $300
    3             ORG   $300
    4  BELL       EQU   $FBDD
    5  START      JSR   BELL       ;RING THE B
ELL
    6  END        RTS
```

NOTE: All listings in this section are shown in standard 40X24 format. However, if tab stops had been zeroed with the TABS command, the 'ELL' of 'RING THE BELL. would have appeared on the same line.

To the right of the column of numbers is the original source code, now formatted. Compare it to the source as originally input. Note that each string of characters has been moved to a specific field. There are four such fields. Counting right from the column of line numbers:

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

Field Two is reserved for opcodes, such as the BIG MAC pseudo-ops OBJ, ORG and EQU, and the 6502 opcodes JSR and RTS.

Field Three is for operands, such as $300, $FBDD and, in this case, BELL.

Field Four will contain any comments.

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

After the line numbers:

Do not space for a label.

Space once after a label (or, if there is no label, once after the line number) for the opcode.

Space once after the opcode for the operand.

Space once after the operand for the comment. If there is no operand, input two spaces after the opcode. The first space after the opcode will cause a tab to the comment field, where a semi-colon [;] is required if no operand is present.

C. SYSTEM AND ENTRY COMMANDS

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

To fully explain and demonstrate each editing option and all system entry commands would require a separate manual. The definitions in Section III, "System Commands" and "Entry Commands", should be sufficient to explain these operations, so 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 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 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 the line number specified in the command.
2. Input an asterisk, and return. Note that INSERT mode has not been exited.
3. Repeat step 2.
4. Input one space, type 'TYA', and return.

On the screen is now the following:

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

5. Hit return, and the system reverts to control mode.

6. LIST the source code.

```
:L

    1 *DEMO PROGRAM 1
    2           OBJ   $300
    3           ORG   $300
    4 BELL      EQU   $FBDD
    5 *
    6 *
    7           TYA
    8 START     JSR   BELL        ;RING THE B
ELL
    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 'D5', and return. Nothing new appears on the screen.
2. LIST the source code. The source listing is one line shorter, one of the asterisk-only lines has disappeared, and the subesquent lines have been re-numbered.

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

1. In control mode, input 'D5,6' and return.
2. LIST the source.

Lines 5 and 6 from the previous example, which contained the remaining asterisk and the TYA opcode, have been deleted, and the subsequent lines renumbered. The listing appears the same as in the subsection on INPUT, Step 13.

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

The EDIT command has 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 and to the right the specified line appears in its formatted state:

6 END RTS

and the cursor is over the 'E' in 'END'.

2. Type Ctrl—D. The character under the cursor disappears. Type Ctrl—D again, and a third time. 'END' 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 'END', and return.
6. LIST the program. Line 6 has been restored.

If you are editing a single line, hitting return restores control mode and the prompt. In step 1 (above), if you had specified a range of lines (ex: 'E3,6') after 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.

The other sub-commands, Ctrl—characters used under the EDIT command, function similarly. Read the definitions in Section III and practice a few operations.

D. ASSEMBLY

The next step in using BIG MAC 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 now the following:

UPDATE SOURCE Y/N?

Type N, hit return, and you will see:

```
]ASM

                        1       *DEMO PROGRAM 1
                        2               OBJ   $300
                        3               ORG   $300
                        4       BELL    EQU   $FBDD
0300- 20 DD FB          5       START   JSR   BELL
      ;RING THE BELL
0303- 60                6       END     RTS

--END ASSEMBLY--

ERRORS: 0

4 BYTES

SYMBOL TABLE - ALPHABETICAL ORDER

    BELL    =$FBDD      ?   END     =$0303
?   START   =$0300

SYMBOL TABLE - NUMERICAL ORDER

?   START   =$0300      ?   END     =$0303
    BELL    =$FBDD
```

If instead of completing the above listing, the system beeps and displays an error message, note the line number referenced in the message, then press return until the ". . . BYTES. . ." message appears. Then refer back to the subsection on INPUT and compare the listing with Step 13. Look especially for elements in incorrect fields.

If all went well, to the right of the column of numbers down the middle of the screen is the, by 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 definitions of OBJ and ORG in Section III — BIG MAC System Pseudo-Ops). After the colon is the number '20'. This is the one-byte hexadecimal code for the opcode JSR.

Note that 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. They are of no use to the computer and, therefore, are not translated into the machine's language.

The next two bytes (each pair of characters is one byte) on line 5 bear a curious resemblance to the last group of characters on line 4; have a look. In line 4 of the source code we told the assembler that the label 'BELL' EQUated with the 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, a 6502 microprocessor convention.

The rest of the information presented explains 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.

E. SAVING AND RUNNING PROGRAMS

The final step in using BIG MAC is running the program.
Before that, it would be a good idea to save the source code.

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 BIG MAC 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 sytem will ask for a file name. Type 'DEMO1', return. After the program has been saved, the prompt returns.
2. Type 'C' (CATALOG) and scan 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 S (SAVE SOURCE FILE) command.
3. Hit any key to return to EXEC mode and input '0', for OBJECT CODE SAVE. The object file is saved under the same name as was earlier specified for the source file. There is no danger of overwriting the source file because no suffix is appended to object code file names.

OBJECT CODE SAVE must be preceded by a successful assembly.

While writing either file to disk, BIG MAC 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, type 'MON', return, and the monitor prompt '*' appears. Input '300G'. Press 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. A control character is always entered by holding down the CTRL key and the character key together.

SUGGESTIONS FOR ADDITIONAL READING may be found on page 8.

SECTION VI GLOSSARY

ABORT —terminate an operation prematurely

ACCESS —locate or retrieve data

ADDRESS —a memory location

ALGORITHM —portion of a program solving a specific problem

ALLOCATE —set aside or reserve space

ASCII —industry standard system of 128 computer codes assigned to specified alphanumeric 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 2 number system, composed solely of the numbers zero and one

BIT —one unit of binary data, either a zero or a one

BRANCH —resume execution at a new location

BUFFER —large temporary data storage area

BYTE —hex representation of 8 binary bits

CARRY —flag in the 6502 status register

CHIP —tiny piece of silicon or germanium containing many integrated circuits

CODE —slang for data or machine language instructions

CTRL —abbrev. 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 or computer program

DECREMENT —decrease value in constant steps

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

DELIMIT —separate, as with a : in a BASIC program line

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

EQUATE —establish a variable

EXPRESSION —actual, implied or symbolic data

FETCH —retrieve or get

FIELD —portion of a data input reserved for a specific type of data

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

HEX —the hexadecimal [BASE 16] number system, composed of the numbers 0-9 and the letters A-F

HIGH ORDER —the first, or most significant byte of a two byte hex address or value

HOOK —vector address to an I/O routine or port

INCREMENT —increase value in constant steps

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

I/O —input/output

INTERFACE —method of interconnecting peripheral equipment

INVERT —change to the opposite state

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

LOOKUP —slang; see table

LOW ORDER —the second, or least significant byte of a two byte hex address or value

LSB —least significant [bit or byte] one with the least value

MACRO —in assemblers, the capability to "call" a subroutine by a symbolic name and place it in the object file

MICROPROCESSOR —heart of a microcomputer, in the Apple, the 6502 chip

MOD —algorithm returning the remainder of a division operation

MODE —particular sub-type of operation

MODULE —portion of a program devoted to a specific function

MNEMONIC —symbolic abbreviation using characters helpful in recalling a function

MSB —most significant [bit or byte] one with the greatest value

NULL —without value

OBJECT CODE —ready to run code produced by an assembler program

OFFSET —value of a displacement

OPCODE —instruction to be executed by the 6502

OPERAND —data to be operated on by a 6502 instruction

PAGE —a 256 byte area of memory named for the first byte of its hex address

PARAMETER —constant or value required by a program or operation to function

PERIPHERAL —external device

POINTER —memory location containing an address to data elsewhere in memory

PORT —physical interconnection point to peripheral equipment

PROMPT —a character asking the user to input data

PSEUDO —artificial, a substitute for

RAM —random access memory

REGISTER —single 6502 or memory location for data storage

RELATIVE —branch made using an offset or displacement

ROM —read only memory

SIGN BIT —bit 7 of a byte; negative if value greater than $80

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

STACK —temporary storage area in RAM used by the 6502 and assembly language programs

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

SWEET 16 —program which simulates a 16 bit microprocessor

SYMBOL —symbolic or mnemonic label

SYNTAX —prescribed method of data entry

TABLE —list of values, words, data referenced by a program

TOGGLE —switch from one state to the other

VARIABLE —alphanumeric expression which may assume or be assigned a number of values

VECTOR —address to be referenced or branched to

SECTION VIII – CREDITS

As a program, BIG MAC is essentially new from the ground up. However, in terms of format, appearance and conventions, it is but one of many improvements and upgradings of the original TED/ASM, as outlined in Section I. Becasue so many individuals have contributed to TED, and hence, BIG MAC, in terms of ideas, programming and documentation, and to this diskette, it would not be cricket to conclude BIG MAC's documentation without expressing our thanks to those listed below for, in *varying degrees,* their efforts. In alphabetical order:

Darrell Aldrich
Ron Aldrich
Allen Baum
Glen Bredon
Dave Garson
Val Golding
Andy Hertzfeld
Neil Konzen
Dan Paymar
Terry Petersen
Pete Rowe
Dick Sedgewick
Gary Shannon
Ken Smith
Peter Soule
Wayne Throop
Steve Wheeler
Randy Wiggington
Steve Wozniak

Without the work and dedication of all of the above, BIG MAC could never have become with we believe it to be: the finest available editor/assembler for the Apple] [.

A special credit is due to Steve Jobs and Steve Wozniak, without whom there never would have been an Apple] [.

SECTION I QUICK REFERENCE COMMAND SUMMARY

Command	Description	Page
	EXEC MODE	
C: CATALOG	Display catalog and allow DOS commands	9
L: LOAD	Load a source file from disk	9
R: READ	Read a text file from disk	LC
S: SAVE	Save a source file to disk	9
W: WRITE	Write a text file to disk	LC
A: APPEND	Load a source file at end of file in memory	9
D: DRIVE	Toggle from drive 1 to drive 2	10
E: EDIT	Enter edit/asm mode	10
Z: ZERO TABS	Enter edit/asm mode with tabs set to 0	10
O: OBJ SAVE	Save object code after successful assembly	10
Q: QUIT	Exit to BASIC	10
	EDITOR	
	Command Mode	
HI*mem:*	Sets upper limit for source file	11
NEW	Deletes present source, resets Himem:	11
PR*#*	Same function as BASIC PR*#*	11
USER	Executes user routine at $3F5	11
TABS	Sets tab stops for editor listing	12
LEN*gth*	Returns number of bytes in source file	12
W*here*	Returns memory address of specified line number	12
MON*itor*	Exits to monitor. Return with Ctrl Y	12
TR*unc*ON	Omits comments prefixed " ;" &ASCII, HEX obj code	12
TR*unc*OF	Reset truncate flag to default	12
STRIP	Strip comments prefixed "*" or " ;" from source	LC
Q*uit*	Exit to EXEC mode	12
SYM	Establishes user symbol table area	LC
ASM	Commences assembly	13
D*elete*	Delete line number, range, or range list	13
R*eplace*	As above, then falls into Insert mode	13
L*ist*	List source file with line numbers	13
P*rint*	List source without line numbers	13
/	Continue List from last line number	14
F*ind*	Find d-string specified	14
C*hange*	Replace d-string1 with d-string 2	14
COPY	Copy line number range to above specified line	14
MOVE	As above, but deletes original lines	14
E*dit*	Edit line number or range specified	14
	Add/insert mode	
A*dd*	Enter text entry mode	15
I*nsert*	Enter text entry mode just above specified line	15
Ctrl L	Case toggle: select opposite case	15
Ctrl O	Enter non-keyboard characters	LC
Ctrl X	Exit text entry mode	15
	Edit mode	
Ctrl I	Insert character(s)	16
Ctrl O	Insert Control character	16
Ctrl D	Delete character(s)	16
Ctrl F	Find next occurrence of character after Ctrl F	16
Ctrl B	Place cursor at beginning of line	16
Ctrl N	Place cursor at end of line	16
Ctrl L	Change case of character under cursor	16
Ctrl R	Restore line to original form	16
Ctrl C	Exit Edit mode	16
Ctrl Q	Enter line up to cursor position	16
[return]	Enter entire line as it appears	16

Command	Description	Page
	ASSEMBLER EXPRESSIONS	
+	Add	18
-	Subtract	18
*	Multiply	18
/	Divide	18
!	Exclusive or	18
.	Or	18
&	And	18
	Decimal data	18
$	Hexadecimal data	18
%	Binary data	18
#	Immediate mode (low byte of expression)	19
#<	Low byte of expression	19
#>	High byte of expression	19
#/	Optional syntax for above	19
	ASSEMBLER PSEUDO-OPS	
	Directives	
EQU	Equate label to address or data	21
=	Alternate syntax for equate	21
VAR	Equate special variables (])	LC
ORG	Establish run address	21
OBJ	Establish alternate assembly storage address	21
SAV	Saves object code to this point.	LC
PUT	Insert file during assembly	LC
CHK	Places obj. code checksum in source	LC
END	Signifies end of source to be assembled	21
	Formatting	
LST ON	Sends assembly to screen or other output	22
LST OFF	Turns off output during assembly	22
EXP ON	Prints all code during assembly	22
EXP OFF	Omits printing macro object code during assembly	22
TR ON	Prints maximum of 3 object bytes during assembly	LC
TR OFF	Resets truncate flag to deafult	LC
PAU	Second pass assembly waits for keypress	22
PAG	Outputs a form feed to printer	22
AST	Outputs n asterisks to assembly listing	22
SKP	Outputs n carriage returns to assembly listing	22
	String	
ASC	Enter a delimited ASCII string into object code	23
DCI	As above, but with hi bit of last chr set opposite	23
INV	Enter a delimited string as inverse	23
FLS	Enter a delimited string as flashing	23
	Data and Allocation	
DA	Enter two byte value, low byte first	24
DDB	As above, but high byte first	24
DFB	Enter multiple bytes of data, delimited by comma	24
HEX	As above, comma not required, hex data only	24
DS	Reserve n bytes of space	24
	Conditionals	
DO	If 0, discontinue assembling code	25
ELSE	Invert the previous assembly condition set by DO	25
FIN	Cancel the last DO	25
	Macros	
MAC	Start of macro definition	26
EOM	End of macro definition	26
<<<	Alternate syntax for above	26
PMC	Assemble macro at present location	26
>>>	Alternate syntax for above	26

LC = Command available on Language Card version only.

Apple
PugetSound
Program
Library
Exchange

304 Main Ave. S.
Suite 300
Renton, Washington
98055
(206) 271-4514