What follows in a partial and unfinished listing of a possible **Director's Cut** of our **Applewriter Cookbook**.

Its file is found **here** and its sourcecode **here**, with the original scanned AWCB **here** for comparison. Or the rare original sometimes **here**.

My director's cuts apply my **Gonzo Utilities** and Acrobat Distiller to fully restore **Linotype era** scanned books and articles to significantly beyond their former glory.

Resulting in amazingly short file sizes (often near 4K per page!), mostly "perfect" error free typography, very "clear" and aligned improved backgrounds, and most artwork newly superb. Plus, of course, optional full URL links and brand new image magnifying clickthrus. And modern nav. All with surprisingly little rekeying.

But at a price of a steep learning curve and being labor intensive.
As well as being inappropriate for legal documents or "Shakespearian" work.

Other Director's cuts and more free eBooks appear **here**.

We can **consult you** on the Director's Cut techniques or do actual reconstruction projects for you.

Listing C.7 summarizes the important entry points. A complete and detailed disassembly script appears in **Listing C.8**. This one will tell you more than you could possibly want to know about every module in the working code.

There's no single answer to the obvious question of "How does Applewriter work?" How you answer depends on what you think is important and where your interests lie. And any attempt to go through the code in numeric order is pretty much fruitless because you lose track of who is doing what to whom.

Instead, let's see whether we can't thread together some of the important working concepts of this program. Our first concern should be the . . .

ProDOS MLI Links

The ProDOS used in ProDOS Applewriter 2.0 is totally stock in every way. The program is also installed in its normal space in high main RAM.

All ProDOS access is by way of its MLI, short for machine language interface, because the klutzy and RAM gobbling BASICS.SYS is not used. To understand ProDOS, you will need the ProDOS Technical Reference Manual and Quality Software's Beneath Apple ProDOS.

Let's see whether we can't give you a few hints. The usual LOAD, STORE, OPEN, etc. commands do not exist when using the MLI. Each time you want to access ProDOS, you do a machine language **JSR $BF00**, immediately followed by a three byte data file.

The first byte gives you the command, and the second two bytes point to a second data file needed to complete the command. The complexity of the file pointed to varies with the command. This second file typically involves less than a dozen bytes.

Everything goes to or comes from ProDOS by way of the **JSR $BF00** MLI interface. On the next page is a ProDOS command summary. More details appear in the various listings and in your tearing of the code itself.

The ProDOS interface is far more uniform and far more flexible than was DOS 3.3e. For instance, the exact same command saves a text file, a BASIC program, or a binary file. The only difference lies in the attributes of the file at the time it is written.

The tab and print constant files are loaded and saved as binary images, putting each in its respective slot in the work files. The glossary and WPL program files are treated similarly, except that they are text files and as before, are loaded into specific places in memory. In fact, the WPL loader does double duty as a glossary loader.

The loader is simply tricked into putting what it reads in the wrong place when loading a glossary. These files are all read to or from main memory.

PRODOS MLI ACCESS COMMANDS

$40 - Allocate interrupt *
$41- Deallocate interrupt *
$65- Quit
$80- Read block *
$81- Write block *
$C0- Create
$C1- Destroy
$C2- Rename
$C3- Set file info
$C4- Get file info
$C5- Volumes on line
$C6- Set prefix
$C7- Get prefix
$C8- Open
$C9- Newline *
$CA- Read
$CB- Write
$CC- Close
$CD- Flush *
$CE- Set mark *
$CF- Get mark
$DO- Set end of file
$D1- Get end of file
$D2- Set buffer *
$03- Get buffer *

*** - Not used by AWD.SYS**

These are also total reads or saves, in which the entire file is loaded or saved at once. Your text files may or may not want to use a total load or a total save.

Applewriter text files often get moved by one 512 byte sector at a time ont or off the disk. This gives an orderly way to search for the delimiters that can allow partial loads and saves.

Moving one sector at a time also solves a memory management hassle because ProDOS will normally load or store into a buffer in main RAM. After searching or processing, all the needed pieces of the loaded or stored text are transferred to auxiliary RAM by the memory management code.

Random access a sector at a time is done with the SET.MARK and READ.MARK commands. Appending is done similarly with the SET.EOF and READ.EOF commands. Generally, a file must be created, opened, read or written to, and finally closed.

A catalog display is done using **GET.FILE.INFO**. This gets handled by routines internal to Applewriter. Files are locked or unlocked by changing the file attributes and then using **SET.FILE.INFO**. Files are deleted with **DESTROY**.

[O] options are unique to ProDOS. **Each ProDOS disk must have a prefix.** Unlike the volume name everyone ignored in DOS 3.3, this prefix must be remembered and available at all times. You also cannot change a disk in a drive without also changing the prefix. A **SET.PREFIX** command exists. As we saw in Chapter 6, a one key glossary entry can greatly ease prefix setting hassles.

The **[O]-F** option is used to list the prefixes of the volumes on line. Because ProDOS has no init code, the volume formatter gets a separate program called **FORMATTER** and installs it from $0800-17FF in main RAM. This code module then gets run, doing an init for you. The same module also completely and destructively overwrites the glossary, your WPL file, and any footnotes in use.

You use the final **[O]-J** option to set Ile modem or printer parameters. What happens is that the baud rate, start and stop bits, parity, etc., are coded in a proper form to set the 6551 serial interface chip in the Ile. The Ile Tecnical Manual gives full details.

Unlike earlier versions of this code, you should have no problems when installing ProDOS Applewriter 2.0 onto virtually any hard disk. This happens because the operating system used is totally standard and the program is completely unlocked and movable.

Nuff said on ProDOS. Let's go on to…

Monitor Access

ProDOS Applewriter 2.0 monitor use is nonexistent..

Applewriter Ile Monitor Use

There ain't any.

ProDOS Applewriter 2.0 uses zero monitor routines. **None at all!** The ROM never gets switched into the high memory area. All of the key getting, disk accessing, and character outputting is done internally to Applewriter. The good news here is the total control you

get with a built in type-ahead buffer, screen lines up to 240 characters wide, controllable scrolling both vertically and horizontally, ease of output routing, and lots more.

The bad news is that some parallel cards on the IIe expect normal use of normal monitor routines. In particular, the IIe does not use location $24, in which many parallel printer cards demand to find some horizontal cursor position information.

As we have seen, **custom patches are needed to handle these problem cards** when using version 2.0. A partial fix is available with the 2.1 update.

Add-on video cards usually will not properly access Applewriter because all screen output characters are handled internally. In fact, these cards are all carefully disconnected by Applewriter as part of the cold startup process in AWD.SYS.

Actually, with horizontal scrolling to 240 characters, nothing special is needed in the way of character display. To further prevent tampering with the internal Applewriter routines, the usual keyboard input hook **KSWH** and **KSWL** ($38 and $39) are set to point to a "brick wall" RTS and then are studiously ignored.

The CSWH and CSWL character output hooks ($36 and $37) are not forgotten. Instead, their primary and only use is to let a serial or parallel interface card adjust them slightly for proper printing.

Ferinstance, if you set this printer hook to $C100 and send a $00 NULL to the interface, the interface will usually reset the hook to $C105 or something similar.

ProDOS Applewriter 2.0 uses this hook only to get the interface card started and set to the right I/0 address. The card then grabs the corrected address for its own internal use.

Characters are output by an internal and protected version of the usual $FDF0 (Fideyfoo) COUT hook. This output is handled by a pointer pair at $9E and $9F on page zero that handles the internal COUT destination setting.

Applewriter's internal COUT can point several possible places . . .

Applewriter Internal COUT ($9E,9F)

1. On a .pd0 print to screen
   Points to $4415 screen code.
2. On a .pd1 or .pd2
   Points to I/0 space as adjusted
   by the interface card or circuit.
3. On a .pd8 print to disk
   Points to $4397 disk write code.

Memory Management

As we have seen, your text files are in auxiliary memory while everything else stays in main memory. Text files are accessed by some code down on page one that does not change as the memory is switched between main RAM and auxiliary RAM. Routines in main RAM that need to read the text file do so via these page one access links.

Remember that the auxiliary RAM text file is really two files. **LOFILE** starts off at $0801 and builds up, and **HIFILE** starts at $BDFE and builds down. $FF markers define the beginning of LOFILE and the end of HIFILE.

The open ends of both files face each other across all the remaining empty space. These open ends are identified with $00 markers. LOFILE holds everything from the start of the message up to **one less** than the current cursor position. HIFILE holds everything from a cursed character to the last character in the file.

Characters are normally entered into the top of LOFILE. All of the characters will be entered as low ASCII, but another routine carefully re-marks each end of each screen line with a high ASCII character instead. Most of the usual routines enter things to the top of LOFILE. Others will pass a character from LOFILE to HIFILE to back up the cursor. Yet others will pass a character from HIFILE to LOFILE to move the cursor forward. [B] and [E] are extreme examples.

Several pointers access the text file. These pointers include LOCURS and HICURS, which point to the open ends of LOFILE and HIFILE. You will also find a screen pointer that starts at a point in LOFILE equal to the top screen line, advances through LOFILE to the cursor, then automatically switches to HIFILE to continue. The screen pointer keys on high ASCII characters to count screen lines.

A printer pointer is used to scan through LOFILE to get characters. Because everything is moved to LOFILE before printing, no switch to HIFILE is needed by this pointer. A general use pointer pair accesses either LOFILE or HIFILE as needed.

Another specialized pointer **($AE,AF)** will back up automatically to get the first high ASCII character that this pointer finds. This character locates the start of any screen line and can be useful for both screen formatting and tabbing.

These pointers all work by switching to read auxiliary RAM, getting a needed value, then immediately switching back to read main RAM. Certain other routines will write to auxiliary RAM by switching to it, doing a store, then switching back to main RAM.

Note that the writing routines can be in main RAM without a conflict. Only the reading routines must be in a portion of the memory that is not switched between main and auxiliary RAM.

Otherwise, as soon as the main-auxiliary switch is flipped, the op codes being read vanish. As we have seen, no switching into the monitor ROM even takes place. Nor is auxiliary page zero or auxiliary high RAM ever activated.

Character Entry

No use is made of the monitor KEYIN routine. If you tried using KEYIN with a different word processor, you probably would drop keystrokes during any hectic typing times.

Instead, ProDOS Applewriter 2.0 uses its own internal routine to get keystrokes. This routine includes a 64 key type-ahead buffer. If your typing gets ahead of the processing, up to 64 keystrokes are saved in a pair of storage buffers.

The main keystrokes are saved to the character buffer at $1D40, and [open apple] and [closed apple] keystrokes are separately saved to the apple buffer at $IFC0 to $1FFF.

Remember that **the apple keys as well as the main keystroke must be saved**, or the computer would not handle certain functions correctly. Two round-and-round pointers keep track of where you are in the key buffer.

A filling pointer $F3 and an emptying pointer $F2 take care of this task.

During non-hectic times, the filler and the emptier stay together, and the keystrokes should get immediately used. At other times, the filler gets ahead, and characters are saved to the buffer. Routines that take lots of time automatically check the keyboard every now and then to make sure nothing gets missed.

A busy signal I (*) prompt appears on the normal status display when busy.

As we saw a while back, characters can still get missed every now and then if a sloppy typist, a bug in the keyboard encoder, and the slower insertion mode all gang up on the key buffer. The buffer seems to be working perfectly when characters are lost. The buffer access is what fouls up the works.

Reviewing, characters can be gotten directly from the keyboard during non-hectic times and otherwise gotten out of the type ahead buffers when things happen too fast.

Several other character sources exist, in addition to the user. Down on page zero is a special WPL and glossary activity flag $0F. Bit #7 or the MSB **N** slot of this flag controls WPL activity, and Bit #6 or the **V** slot controls glossary activity.

If the glossary is active, the character is gotten from the glossary file. Similarly, if WPL is active, the character is gotten from the WPL program file.

Sometimes the WPL file will involve itself with its $A-$D strings. If WPL and these strings are active, the $A-$D string becomes the source for the next character to be used. You'll find a separate string activity flag at $F6 to handle $A-$D activity.

Sometimes you want to use a string already in the machine, such as the **=** filename or something else that has been previously formatted or put together. A special string flag

$AD exists for such cases. If this string flag is set, the old string, which is usually in the key buffer at $0200, is used one character at a time. If the string flag is cleared, new characters are gotten from the user, the type-ahead buffer, the glossary, WPL, or the $A-D flags.

Yet another source for strings of characters exists. When doing a [Q]-I, you can receive its characters directly from a modem or by way of a modem buffer that scan dave incoming characters during hectic times. This access bypasses the usual key-getting routines.

ProDOS activities, such as loads and stores, completely bypass any key getting routines and usually put their values directly where they belong. If searching for delimiters is needed, it is done one 512 byte sector at a time by way of a user buffer at $8900.

The majority of the word processor's time consists of patiently waiting for the user to input a new keystroke. Regardless of a keystroke's source, after that keystroke is received, it gets filtered for control and cursor motion commands. If a valid command is found, it is carried out. If not, the character is entered to the top of LOFILE.

Summarizing . . .

SOURCES OF KEYSTROKES

1. Directly from the user during non-hectic times.
2. Indirectly from the user via a type-ahead buffer.
3. From the glossary during glossary activity.
4. From the WPL program during its active use.
5. From the $A-$D strings in WPL if used.
6. From an old string already in the keybuffer.

We have seen that several sources of keystrokes are available, all of which can be handled internally by the code. User input is accepted directly or is stashed in a pair of buffers if the processor is busy. Characters can also come from the glossary, from WPL, or from a WPL $A-$D string if the controlling flags are set properly.

Sometimes, an old string will be reused instead of getting new input. And finally, all your characters can come directly from the modem or by way of its type-ahead buffer, and thus bypassing the usual key getting routines.

Now for some details on the …

SCREEN DISPLAY

The screen display has some very sneaky and complicated code associated with it. First note that **you can turn the screen off and on with flag $F7**. Leaving the screen off speeds up WPL operation considerably. Naturally, seeing what you are doing when the screen is off is tricky. A screen that is turned off is useful, though, to display WPL menus, prompts, and a few other operations.

Before a screen display is updated, any routine that messes with the text files will reformat the screen lines in that file. Reformatting is done by backing up two lines from the cursed position and then counting how many whole words will fit on a line. Each line stops either on a carriage return or when the line does not have enough room for the next word.

At that point, a marker character, usually an $0D carriage return or an $20 space, will get changed to a high ASCII $8D or $A0 and restored to the text file. All older and lower ASCII characters are erased from the text file. The process continues forward through the text file until a carriage return is found that is already correctly formatted.

Note that anything two lines before the current activity had to be correct already, thanks to previous reformatting. Everything beyond the next carriage return is also correct. Only the mess in the middle needs straightening out. The entire text file is reformatted after a margin altering [A], after loading, after printing, and any other time that something really major happens.

Completely reformatting a long text file may take you several seconds. The upshot is that, before a screen update, all of LOFILE and all of HIFILE have end of screen line markers that are properly placed to end each line on a whole word.

The cursor usually stays on the middle line of the active screen. Should the screen overflow, everything will scroll up one line. Should it underflow, everything will back down one line. During insertions, characters get turnstiled as far as they have to in order to reach the next carriage return. To update the full screen, the screen pointer pair $88,89 backs up 12 lines, which is usually 12 inverse ASCII characters from the top of LOFILE.

Characters are removed from LOFILE and put on the screen up to your cursed location. Immediately beyond LOCURS, the pointer is moved to HICURS, and the code continues filling in characters from HIFILE until 12 more lines are completed.

The flashing you see on the cursed character is purely your imagination at work. For the service routine that awaits a keystroke patiently flips the cursed character on the screen between low and high ASCII. Sometimes that character is left as an inverse low-ASCII marker. An example is the cursor on the nonactive side of the split screen.

Note that **the large and empty no man's land between LOCURS and HICURS is bypassed**. The lowest character in HIFILE ends up at the cursed location. Note also that the alternate character set will get used here, which has no flashing characters available. Low ASCII characters appear as inverse text.

Only the active half of the screen is updated on a split screen. The inactive half of the screen remains static, remembering things the way they were.

If the wraparound flag $El is not active, characters will get put on the screen wall to wall without regard for word breaks. Only a 79 character line gets used because room must be left for the optional column 80 carriage return display.

A user prompt is sometimes needed at the bottom of the active screen. To print a prompt on the screen, three lines are erased, and the prompt is placed on the middle line. Prompts are normally read as needed out of the reference file area. Service subs are built into the screen code for the live cursor screen motions, line motions, line clearing, and scrolling.

Another summary…

SCREEN UPDATES

1. Before any screen update, low ASCII markers are placed at the end of each text file screen.
2. Everything before the cursor on the screen comes from **LOFILE**.
3. The cursed character and everything beyond comes from **HIFILE**.

Things get more complex if you are using a right margin wider than 78 columns.

In this case, not all of the screen line can be displayed. A special stash is used to calculate the offset needed between the previous end of screen line marker and the actual screen starting text character. This offset is automatically added when finding text file characters to go on the screen.

As long as the cursor stays near the middle of the screen, no change is made in the offset. If the cursor gets left of the twelfth character, offset is decremented, giving an apparent horizontal scrolling of one character to the left. Should the cursor end up right of the sixty eighth character, the offset is incremented, giving you an apparent horizontal scrolling of one character to the right.

The display will start with its left margin LM value in column zero, which will produce an apparent what-you-see-is-what-you-get display, as long as the screen width is less than 78 characters total. For the most exact display, be sure to use **[tab]/tx rather than PM values for all of your paragraphs**. Note that all screen lines will be justified flush left even if a wide left margin is used.

Note also that breaks on whole words only are required on lines wider than 78 characters.

We now know something about how ProDOS works, how the monitor can be used, where the characters come from, how they are managed, and just how the screen update works.

Next are the…

Individual Control Commands

Let's run down the control command list, seeing roughly what each command does. For more detail, check Listing C.8 or your own disassembly listing and cross reference list.

[@] is really a **[delete]**, recoded to $80 from its default value of $FF. This command will unconditionally knock out LOFILE's uppermost character and replacing it by using a $00 marker. The command then backs LOCURS up one character.

[A] is the command to alter the screen margins. If the characters per line are less than 78, the left screen margin is set to appear in column zero. If the characters per line are more than 78, the left screen line is first set to center the cursor if possible. As the cursor gets moved within 12 characters of either the left or right margin, horizontal scrolling will be activated. Screen lines are marked by setting the last character to high ASCII.

[B] moves all the characters from LOFILE to HIFILE, placing the cursor at the beginning of the text. When finished, LOFILE will be completely empty, and HIFILE will hold the text being processed.

[C] changes the case flag, initially from none to U or later from U to L or back from L to U. When characters are entered, this flag is checked. If active, uppercase or lowercase is forced as chosen. The flag is reset on all cursor motions except the left and right arrows. These arrows let you capitalize or lowercase as many characters in a row as you want. Only real letters are changed.

[D] toggles the data direction flag between < and > . If a [W] or [X] is specified with a data direction of >, words or paragraphs are restored. If < is the data direction when [W] or [X] are specified, words or paragraphs are deleted. The data direction flag also will set the direction of a search or search and replace.

[E] moves all the characters from HIFILE to LOFILE, placing the cursor at the end of the text. When completed, HIFILE is completely empty, and LOFILE holds all of the text.

[F] does either a search or a search and replace. Delimiters are interpreted, substituting special ones if used. Then the text is searched using the $98, 99 pointer pair. If you want to make a replacement, text is moved from HIFILE to a work buffer and the replacement is made. Various options substitute for fake carriage returns, allow repeats for all occurrences, let you use wild cards, and provide any length capabilities

[G] either sets up or reads the glossary. If a valid read, the glossary flag is set. If set, characters are gotten from the glossary work file until the next carriage return. At that time, the glossary flag is cleared. If the flag is a *, the glossary is emptied by placing a zero at the glossary start location $1BOO. If the flag is a ?, the end of the glossary is found and the new definition is entered that ends with a carriage return and a $00. The glossary has a nest that works like a subroutine and remembers up to eight of the return pointers. This nesting picks back up on the caller when the callee is finished.

[H] is the left arrow. When it is the only key pressed, it backs up one location by moving one character from LOFILE to HIFILE. When used with [closed apple], the left arrow (H) does an express-by-word backspace, continually backing up until the first space is found. With [open apple], the left arrow saves a character to the swallow buffer instead of HIFILE and increments the round and round swallow buffer pointer $AC.

[I] moves the cursor to a tab. The present position since the last carriage return is first calculated. A test then gets made to see whether any valid tabs exist beyond the present position. If so, spaces are added to the top of LOFILE to move to the next tab position. If [closed apple] happens to be down, the cursor is moved without space padding so that the characters are tabbed over without being moved. Tabs are permitted anywhere.

[J] is the down arrow. When it is the only key pressed, it moves characters from HIFILE to LOFILE, repeatedly frontspacing until one line is moved. Each succeeding line ends with a high ASCII marker. With [closed apple] and if enough text is left, the down arrow goes forward 12 whole lines.

[K] is the up arrow. It moves characters from LOFILE to HIFILE, repeatedly backspacing until one line is moved. Each preceding line ends with a high ASCII marker. With [closed apple], the up arrow tries to go backward 12 whole lines if enough text is available.

[L] is the load command. Loading can be from the text file, which is really a copy command, or from ProDOS. Loading from ProDOS is first done via a one sector, 5 1 2 byte buffer at $B900 in main RAM. After scanning for any needed delimiters, the characters are transferred to the top of the LOFILE text file area in auxiliary RAM. Text is entered just beyond the present screen position. Alternate delimiters provide for all occurrences, wild cards, and fake carriage returns. An option exists to load only to screen.

[M] is the carriage return that ends each command. This command is not available for other uses, although you can fake a glossary carriage return with a) and a search for a carriage return with a special delimiter, such as >.

[N] is the new command. Because this command can be deadly, you are given a prompt that needs a Y answer. If you are serious about destroying your text file, this command adjusts the HIFILE and LOFILE pointers so that nothing is in either HIFILE or LOFILE and your cursor is sitting at the beginning of LOFILE. The old material is not erased, except for the first character. All that happens is that the first character gets replaced with an open-end-of-file $00 marker.

[O] is the DOS access menu. The menu is displayed and a selection is gotten. On a catalog command, a GET. FILE. INFO is done for the directory. The catalog formatting is internal to Applewriter. Locking and unlocking are done by reading, then changing the attributes of a file. Renaming, deleting, setting prefixes, finding volumes on line, or making a subdirectory are done directly with their respective ProDOS commands. Initing a new disk is done by loading a separate formatting program, then jumping to that program. The formatter destructively overwrites the glossary, WPL, and any footnotes. The printer commands are a set of internal routines that let you set the baud rate, word length, stop bits, and parity on a Ile. This routine also defeats video echo and suppresses any carriage returns that may be generated by the interface hardware.

[P] updates the print/program file or carries out a WPL command. A valid two-character, print/program value is converted to hex and entered in the correct slot in the file. Absolute values are entered as such. Relative values are added to or subtracted from the old value. Two's complementing is used for subtraction. On TL and BL entries, the string is placed in the correct file. On UT, the underline token is saved. On NP, CP, and WPL commands, the selected command is completed. [Q] accesses the additional functions menu. Binary tab and print/program values are loaded or saved as called for, using the ProDOS MLI. All of these values go in their respective stashes in main memory.

[Q] accesses the additional functions menu. Binary tab and print/program values will get loaded or saved as called for, using the ProDOS MLI. All of these values go in their stashes in main memory. Glossary or WPL loads and saves are done similarly. The carriage return toggle sets or clears a display flag. The status toggle is identical to [esc] and may be replaced with something useful. Connecting printer to modem gives you a limited way to type directly to your printer. More importantly and more usefully, this selection also lets you send or receive text files over a modem. A submenu on the [Q]-I selection lets you activate these modem features, such as recording incoming modem data or filtering control commands. The Quit option provides for an orderly exit to some other ProDOS system application program. Quitting includes reconnecting all disconnected video cards, and closing out current ProDOS activity in an orderly way.

[R] toggles the replace mode flag $F5. When in the replace mode, a character is deleted from HIFILE before each character entry, then the new character is entered into the top of LOFILE as usual. The combination of deleting the cursed character and entering another character at the cursed position gives the illusion of replacing the old character. Replace mode is aborted on most cursor motions.

[S] is the save command. On any save, the entire text is first moved to LOFILE. Then all or delimited portions of the text are moved to a sector buffer in main RAM at $B700. Full sectors are transferred to disk as they are filled. Should appending be needed, ProDOS markers are set to allow adding to the end of an existing file rather than overwriting.

[T] sets or clears tabs. On a purge, the entire tab file is cleared to all zeros. On a Clear, only one pair of tab entries is set to zero. On a Set, 64 tabs are allowed. A tab status display is updated, causing set tabs to appear inverse and all cleared tabs to appear normal. Although the status display only goes to 240 columns, tabs themselves can exceed this number.

[U] is the right arrow or frontspace. When it is the only key pressed, it moves the cursor forward one location by moving one character from HIFILE to LOFILE. With [closed apple}, the right arrow does an express-by-word frontspace, continually going forward until the first space is found. With [open apple], the right arrow retrieves a character from the swallow buffer instead of from HIFILE, placing the character in the top of LOFILE, and decrements the round and round swallow buffer pointer $AC.

[V] toggles the verbatim flag $72. With this flag set, all control characters except [M] or [V] are entered directly into the text file. This allows imbedded control characters for such things as special printing or typesetting commands. With the V flag cleared, control characters are used in their normal manner.

[W] inserts or deletes a whole word. On < , a word is saved to the word and paragraph deletion buffer starting at the first open spot available. Characters are removed from the top of LOFILE and placed into this buffer until either a space or an empty file is found. On > , a word is recovered from the word deletion buffer, putting the characters in the top of LOFILE and stopping on a space. A round and round pointer pair $94,95 keeps track of positions in the deletion buffer. A deletion counter prevents buffer overflow.

[X] is similar to [W] but [X] inserts or deletes an entire paragraph, keying on a carriage return rather than a space. On [W] and [X], if [closed apple] is used, the word or paragraph is saved to file but not deleted. This is most useful for copying short blocks of text.

[Y] is the screen splitting switch. On a [Y], the split screen is set up, using only 12 lines per display rather than the usual 24. One side of the split screen is active at a time. The other side is a static display of the way things were. Pointer $F8 decides which side is active. On a [Y] with a split screen, control flips over to the other screen side by toggling $F8. On a [Y], the pointer is cleared, allowing the normal full screen display.

[Z] toggles the wraparound flag at $EI. Wraparound is always present in the text file since each screen line ends with a high ASCII marker. If this flag is active, the screen update code ends each line on these markers. If wraparound is not necessary, characters are put on screen as they occur, stopping at 79 screen characters. The character slot to the extreme right is always reserved for a possible carriage return symbol, whether or not it is used. Note that full word breaks must be used if more than 80 columns are active.

[_] calculates the page/position display. This routine is cumbersome and slow but also is most useful. Because operation is too slow for real time, you must toggle [_] only when you want specific page/position information. The routine works by counting carriage returns and comparing them to the printable lines per page. The total carriage returns are divided by the printable lines per page. The result gives you the page, and the remainder gives you the position on the final page. The need for division causes the slowness.

We aren't quite through with control commands because I have saved two of the heavies for last. As a reminder, we are scanning through the various features of this program to see roughly what they do. Much more detail is found in Listing C.8 and in your own torn disassembly and cross reference. Our first heavy is …

Printing

Applewriter 2.0 printing routines are part of the machine-resident editing code rather than a separately loaded disk module. In Applewriter, you have a choice of four possible print destinations. You can print to a real printer to get a hard copy. Or print to a modem or a special Ile plug-in card. You can print to the screen to see exactly what your printed text will look like, or you can print directly to a disk text file.

The last option gives you a document in final form, without any imbedded commands, that looks exactly like the document to be sent to the printer. Printing to pd8 is particularly useful when you are typesetting, need camera-ready copy, require multiple columns, want multiline headers or footers, or are transmitting between two different computer brands. If anything seems like it cannot be done with Applewriter, chances are that a trip through pd8 land will bail you out one way or another. Once you decide what you want, WPL can make results invisible and automatic.

One gotcha: Be sure to have a unique filename for your pd8 images! Otherwise pd8 files will get mixed up with your files that contain embedded commands and will royally foul the works. I often use a generic ZZZ for any temporary use of a pd8 file. Print destination is specified with the pd command. A pd0 outputs to the screen for WYSIWYG previews. A pdl dumps to a printer card in the selected slot. Rarely a pd2 or pd4 could be used to dump to a modem or some other special card. A pd8 dumps directly to the disk.

Printing begins by moving everything to LOFILE with a [E] command. The printing pointer pair $90,91 then moves up through the text file by starting at $0801 and grabbing one character at a time. Pages are formatted using the print/program values, such as top margin, left margin, right margin, bottom margin, page numbers, etc. At the beginning of the first page, the pn page number is saved to the running page counter pair at $BE,BF. The default left and right margins are saved as well. This way, the top and bottom line formats will stay the same throughout the document. The top line, if used, is formatted and printed first. This is done by reading three delimited pieces out of the top line file and then moving them to a work area where the page number can be substituted.

Each left, center, or right piece is moved to a line buffer that has been previously filled to all spaces. The left piece starts at the left. The center piece starts half way across minus half the length of the center text. The right piece begins shy of the right margin by its length. After the top line, the top margin padding is put down, followed by the body of the page. The body is formatted and printed one line at a time, allowing for paragraph margins or outdents on the first line in each paragraph. Each line begins by getting enough characters out of the text file to fill the line.

As the characters come in, they are filtered for imbedded commands and for footnotes. lmbedded commands start with a carriage return followed by a period followed by two or more letters. If these commands are found, the printing will stop long enough to let the imbedded command do its thing. For instance, on an .lm + 5 command, printing halts momentarily. The left margin is retrieved, decimal five is added to it and then the left margin is replaced. The new left margin value will be picked up on the next line.

Any command that Applewriter does not recognize is treated as printable characters. This leads to the shortline problem. We have seen a STRETCHIFIER patch described in Chapter 6 that cures this hassle. Characters are also filtered for footnotes, which begin with the l< command. If footnotes are found, they are stored in the footnote buffer at $1400, and the footnote flag $FE is set. This flag is incremented once for each footnote.

The very first footnote knocks two counts off the available number of printed lines. Any additional footnotes knock off one extra line. This gives a space between the bottom body line and the first footnote line. At print time, any user separators (_) are automatically converted to NULL commands. That conversion works fine if you need NULLs for an old Epson. It is terrible if you need a user separator for a daisywheel HMI command, a modem activity command, or for expanded printing on some newer dot matrix printers. A fix for this is described in Chapter 6.

At any rate, characters are gotten and filtered until enough whole words are entered to fit between the left and right margins. These characters are placed into your line formatting buffer at $IC00. That line is then justified. Should left justification be in use, nothing more is done. All of the words remain flush left.

If center justification is in use, the length of the entered characters is subtracted from the line width. This new length is halved and then that number of spaces is used to off set the characters in the line buffer. If right justification is in use, the length of the character string is subtracted from the line width, and that number offsets the characters in the line buffer. In any of these three modes, you end up with the buffer holding the line justified in the correct position.

Spaces are added as needed before the center justified and right justified text. Spaces are not needed beyond any text because the carriage return completes your entry. A row of printed spaces looks the same as the unprinted page, so trailing spaces are not needed.

On the fill justification of a long line, the needed number of padding spaces is calculated. Text is then moved one space to the right, beginning with the first space and repeating as often as needed to force the fill justification.

Microjustification is not available inside stock Applewriter. Instead you can use imbedded commands to tell an intelligent printer to microjustify for you. Naturally, if your printer has full microjustification available internally, your text will look much better than text justified by whole spaces. As we've seen, the enhanced Diablo 630 microjustifies beautifully.

Regardless of the justification mode, all of the characters end in the correct place in the line justification buffer. When the justified line is output for printing, it is preceded by enough spaces to make the left margin. On first paragraph lines, the pm value is used to adjust the needed number of leading spaces.

As the line is printed, the characters are filtered for the underline token. Should this token appear, it is replaced with a space, and an underline mode flag $E0 is toggled. Underlining is done by printing the underline character and then backing up one space and printing the character to be underlined. Such underlining will not work on certain very old or otherwise primitive dot matrix printers. **The printer must be able to recognize the $88 ASCII backspace command for this type of underlining.**

As we have seen, **underline is best left to the printer**. This is done by imbedding suitable commands to turn the printer's underliner on and off when needed. As many lines as are asked for are put in the body of the text. When finished, any footnotes are recovered from the footnote buffer and printed.

They are followed by the bottom line padding, and, if used, the bottom line. Note that the stock program allows only a single top or bottom line. However, with some repeated trips through **.pd8** land, you can have any number of top and bottom lines. You can also single space the headers and footers while double spacing your main text, as well as using even or odd headers.

Good old **.pd8** will also let you do space-and-a-half and similar tricks. Printing continues until all of LOFILE has been printed. At that point, a new file can be loaded and a new cp continue printing command can be given, picking up exactly where you left off.

The same running page number and current margin settings are kept. On the single sheet option, printing halts at the bottom of the page long enough for you to change paper.

By the way, if your Ile printer card does not defeat video echo, it will trash the screen and might slow things down, particularly at higher serial baud rates. Your Ile serial interface automatically defeats any screen echo when you set the printer interface with [O]-J. As a reminder, special patches may be needed for intelligent Ile printing cards.

You will, of course, get the best printing with an intelligent printer or a typesetter that accepts imbedded commands and can do its own proportional spacing, boldface, italics, shadow printing, and microjustification.

So much for printing. The real biggie is …

WPL

WPL is a supervisory language that looks like a cross between PASCAL and assembler. Its intended use is as an executive controller that will handle long and involved tasks for you. Obvious uses are printing a multiple file book chapter having the correct headings and footings, customizing a mailing to a separate address list, counting words, putting down menus, prompting operators, building an index, etc. But it's the non-obvious uses of WPL that boggle the mind.

The amazing thing about WPL is how much is done with how little. The additional code needed is rather short and compact. I have used WPL to insert or remove the line numbers from assembly code and to picture process strings sent to a plotter.

I have used WPL to trick a printer into doing camera-ready copy and to handle automatic formatting. I have also used WPL to create high level graphic images. I am convinced that **WPL is far more powerful at processing pictures** than it is at words. Others have even written adventures in WPL.

WPL interfaces beautifully with Postscript, the typesetting language that is used on the Laserwriter. We already saw how to use WPL to completely format a document for full bells and whistles superior quality printing.

The message is whelming. WPL is super powerful and important. Without this language, Applewriter may have some second rate competition. With WPL, that's all she wrote…

If you do not both thoroughly know and aggressively use WPL, you are passing up up at least 98 percent of all of the good stuff you can do with Applewriter!

And your hidden secret message is…

So get with it. Now!

Figure 7.4 summarizes a WPL instruction. Each WPL instruction is one line long and ends with a carriage return. Lines are done normally in the order they are found in a WPL program although several important exceptions exist.

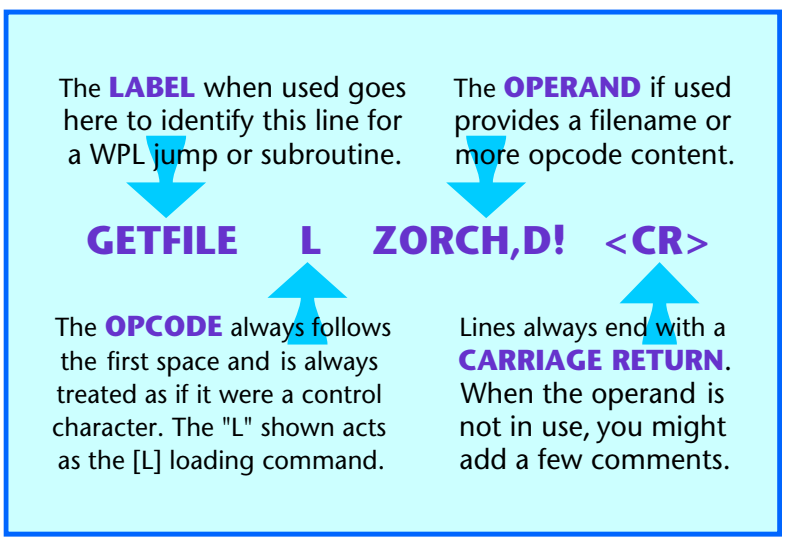

Fig. 7.4. A WPL command line is similar to assembly source code.

Each WPL line may begin with a label. The label must not have any spaces. If a label is used, it lets WPL find a certain line for possible jump or subroutine access.

If a label is not used, **a space must be the first character on a WPL line**. Either way, the first character after the first space in a WPL line gets treated as if that character were a

control character. WPL then behaves just like you typed that control character from the keyboard. For instance, the WPL interpreter would see a line that is made up of a space followed by a B as a **[B]** and would move the cursor to the beginning of the screen.

Although WPL lets you use lots of spaces for pretty printing, you can run out of program room real fast if you try this. Thus, most non-trivial WPL programs are usually done in a compact and hard to read form. Nearly anything you can do at the keyboard, WPL can do for you, automatically, potently, and without errors.

You can think of WPL as a high level language that is extremely good at editing any long strings of characters and acting on them plus being a disk and printer supervisor.

So what is WPL and how does it work? To answer, we will first need a way to write a WPL program. Because **a WPL program is nothing but some processed words**, you write your WPL program on Applewriter, just like any old text file, and save it to disk.

One WPL command is called **do**. To run your WPL program ZORCH, you simply enter **[P] do ZORCH**. That is all there is to it. The do code first clears all the various WPL flags and work areas. This code then loads the named program into a WPL program file starting at $1000. WPL length can be 1024 characters with footnotes or 2048 characters without.

Note that **you can beat the 1K or 2K character limit so that your WPL program can end up arbitrarily long** You chain any number of WPL programs together end to end with do commands.

You can also use one main WPL supervisory program to control several others. The others return back to the supervisor after carefully setting a variable or two to tell the supervisor where the program left off. Variables are preserved whenever WPL programs are chained or otherwise linked together.

The do command also sets the WPL activity flag $DF so that keystrokes will be read from the WPL file rather than from the keyboard. **When the WPL flag is set, the first line of the WPL program is read**. If a label is present, it is passed over, and WPL finds the first character beyond the first space or string of spaces.

This character is converted into a control command and gets processed just the way any control characters entered from the keyboard would be. Any remaining characters on the line are used as needed by the control command. Ferinstance, a filename might follow an L for **[L]**oad, but a search and replace string might follow a F for **[F]**ind.

The WPL lines are read one at a time, usually in sequential order. Each line terminates with a carriage return. Your final WPL line ends with the $00 marker, which stops WPL and returns control to the keyboard. WPL has jumps and subroutines. The WPL command go will start at the beginning of the WPL file and search for a label. On the jump command, that label is found and the program unconditionally jumps to that line and then continues. The WPL command **sr** does almost the same thing for subroutine access. The only major difference is that a return address is remembered on a WPL stack at $1D00, along with a

stack pointer $92 that remembers where to return to. Returning is done whenever a RS command is found. Subroutines can be nested to a depth of 32.

WPL has three numeric variables named (x), (y), and (z). Each could range from 0 to 65535. Any time an (x) is found, the value assigned to (x) will be substituted, and the same goes for (y) or (z). You can set these numerics to any value, either absolute or relative.

You can easily test a numeric for zero. With some hassle, you can also test a numeric for most any nonzero value. For instance, psx45 places a decimal 45 into (x). psx7 sets (x) unconditionally to decimal seven. A command of psx + 7 adds seven to whatever was already in (x).

Most importantly, the command psx-1 decrements a counter loop involving (x) by a single count. The numerics are really nothing but print/program values and are stashed in the print/program file, such as Im or ut. See Listing C.3 for the exact locations.

Substitutions are done at the time the WPL line is interpreted. WPL has string variables. Four of them are named $A through $D. These are stashed in the work files that start at $1E00. Just like the numerics, the strings are substituted for their symbols at the time the WPL is interpreted.

Strings may be loaded from memory or disk with the **ls** command that is assigned to an immediate value with the as command and compared with the **cs** command. You can check Listings C.3 and C.8 for more details.

During disk access, the **is** load string command borrows an unused portion of the text file immediately above LOCURS out in no man's land. Because the $A-D strings are allowed to be at most 64 characters long, there is little danger of crashing into HICURS, except on a nearly full text file.

String loader is done in the text file run to give all of the powerful loading options to WPL strings that the usual text loads receive. After use, the string above LOCURS is zeroed out so that this string does not end up becoming an unwanted part of your text file.

WPL has conditional execution. This is an absolutely essential feature of any computer language. The next WPL statement is skipped if a numeric reaches zero, if [F]ind cannot, if [L]oad will not, or if sc does not compare. The skipped statement is usually a jump, or a subroutine call, or a program quit. Thus you can make a test and cause WPL to pick two different routes, depending on the result of that test.

WPL interacts with the user. You can clear the screen or print fixed screen messages with your **ppr** command. You can get a string from the user with a pin command. The display can be turned on with the pyd command and off with the pnd command. An off display computes much faster, besides holding the last prompt or message for you.

A pep command in WPL enables the printer if its value is not zero. You use this command to print only the page you want in the middle of a document. To accomplish this, put an **.ep0** at the start of your document and an **.epl** where you want the printing to start.

Listing C.9-cont...

```
$3358-33A9 -- UPDATE TAB STATUS IMAGE

    Fill the tab image $B600-B6FF with all dots. Write a 1,
    2, 3… etc to each tenth position, using the 6502 in its
    DECIMAL(!) mode. Write a suitable "fives" marker to each
    five slot, with a single quote for 5-95, an exclamation
    point for 105-195, and vertical bar for 205-235. Mark set
    tabs by scanning the tab address file. Complement the
    character in the tab image for each set tab.

$33AA-33B2 -- INIT POINTERS AND FLAGS

    Init LOFILE. Init HIFILE. Adjust screen margins. Fall
    through to next module.

$33B3-33C5 -- INIT FLAGS

    Set for normal data line. Use wraparound. Update entire
    screen; no bottom prompt. Do not display page/position.
    Set data direction to "<". Reset case and verbatum flags.

$33C6-344E -- MAIN WORD PROCESSING ENTRY

    Reset stack. Init flags. Reset busy flag. Turn screen on.
    Update HICURS pointer. Update LOCURS pointer.
    Reformat screen margins. Unsplit screen. If startup flag
    is set, then get and run startup WPL routine, and update
    print values. Either way, fall through to next module.

$33Fl-344E -- MAIN WORD PROCESSING SERVICE LOOP

    Set $FDmemory load flag. Set not copy $77 flag to not copy
    Clear load string flag $79 to allow user input. Clear $71
    main memory source. If type ahead buffer is not empty, and
    if not slave modem mode, then update the screen. If WPL is
    not active, and if string flag $AD not set, then reset the
    string flag, get user response, and redo the loop. Read the
    keyboard. If an [esc] and if WPL is active and if not in a
    WPL subroutine, then quit WPL and  repeat the main service
    loop.  Get user character. Get help if open-apple or ?.
    Reject all  NULL $00 characters. If a delete key, do the
    deletion. Otherwise, process character and repeat loop.

$344F-348D -- FILTER CHARACTER

    If WPL but not glossary, process character as if it were a
    control command. Bypass control testing if [V]. Force low
    ASCII and process as control command if control command.
    Reset case flag. Process carriage return as control command.
    Reset the page/position display flag $78. Send the character
    to both file and screen. Reformat screen markers.
```

Listing C.9-cont...

$348E-34C3 -- FILTER CONTROL COMMAND

Force control code and hold locally to $C2. Test reformat flag $CD and reformat screen if set. If an [esc] key, toggle the data display and exit. Turn replace flag $F5 off unless [R]. Zero mystery flag $ED. Set main utility pointer to start of control command prompt file. Scan the control prompt file, going one past each "[" seeking a match to current control character. If a match is found, remember twice its position in the list to the X register. If no match found, quit on end $00 marker. Use the X register position pointer to pick an address pair. Shove this pair on thestack, and do an RTS, using the forced sub return method to do an indirect jump to the intended control function. Note that the jump goes to the forced address **PLUS ONE**.

$34F9-3528 -- SPLIT SCREEN SETUP

If WPL is not active, clear the bottom of the screen and put down the (Y] user prompt. Get user response and clear prompt. Forceupper case. If a "Y", turn split screens on.If a "N", turn split screens off. If a carriage return, swap screens only if [Y] is already active.

$3529-3533 -- SWITCH TO OTHER SPLIT SCREEN

Abort if [Y] is not active. Switch screens by changing the "V" slot of $F8 from one to zero, or vice versa. Jump to screen pointer fixer.

$3534-353F -- TURN SPLIT SCREEN ON

Force [Y) flag to split screen on, lowerscreen active. Set the split screen pointer $F9,FA to present LOCURS position Jump to screen pointer fixer.

$3543-354E -- TURN SPLIT SCREEN OFF

Abort if WPL is active. Reset split screen flag to $00 for one full screen. Jump to recalculate the vertical screen.

$354F-357D -- FIX SPLIT SCREEN POINTERS

Update the screen. Exit if split screen is off. Check which split screen is active, and calculate VPOS position, using $00 for the top screen and $QC for the bottom screen, twelve lines down. Save the static cursor position to $98,99. Save present LOCURS cursor position to $F9,FA. Move characters from HIFILE to LOFILE or vice versa as needed to get to the desired point in the file.Init LOFILE on underflow.

Listing C.9-cont...

```
$357E-3593 -- CALCULATE SCREEN WINDOW

    If upper split screen, set window top at $00 and window
    bottom at $0C. If lower split screen, use $0C and $18. If
    full screen use $00 and $18. Set horizontal cursor to left.
    BASH tha vertical screen address.

$3594-359£ -- TOGGLE DATA LINE DISPLAY

    Advance the $ES data display flag to its next of three
    possible values, with $00 being no display, $80 being the
    usual HEM-LEN-POS display, and $CO being the tab display.

$359F-35AD -- TOGGLE WRAPAROUND MODE

    Abort with ding dong if the screen right margin is not set
    to 80 characters, since broken words are only allowed on a
    full screen display. If a full screen display, change the
    $El wraparound flag either from or to its $00 whole words,
     or its  $FF broken words setting.

$35AE-35BS -- TOGGLE CARRIAGE RETURN DISPLAY

    Change the $74 carriage return flag from or to its $00
    normal display or $FF show returns as an inverse "M"..

$35B6-35BC -- TOGGLE VERBATIUM FLAG

    Change the $72 [V]erbatium flag from or to its $00 normal
    use or its $FF imbed control characters directly.

$35B0-35C2 -- PROMPT SCREEN BOTTOM ANO GET RESPONSE

    Short code link to first prompt the screen bottom and then
    get the user response.

$35C3-3508 -- FORMAT FILENAME ANO PRINT TO SCREEN

    Clear and prompt screen bottom. Print the old filename to
    the  screen. Get any user changes to the filename. Abort on
    a "'?"  for catalog or a carriage return. If the "="
    filename is  to be used, scan the old filename to the first
    comma, and then copy the keybuffer beyond the comma. If a
    new filename, transfer the entire filename. Zero out the rest
    of the filename buffer. Bold the filename length to $E9.

$360B-3623 -- CLEAR ANO PROMPT SCREEN BOTTOM

    Abort if WPL is active. Clear bottom of screen. Print the
    selected control command prompt to screen, stopping on the
    first space. Then print a "•" to the screen.
```

Listing C.9-cont...

```
$3624-3637 -- SET CURSOR FOR BOTTOM WINDOW

    Set HPOS to zero. Set VPOS to 9 lines from window top if
    split screen, or 21 lines from window top if full screen.
    Make room for three lines, usually a blank line, a prompt,
    and a second blank line. BASH VPOS.

$3638-3646 -- CLEAR BOTTOM OF SCREEN

    Set cursor for window bottom. Clear window if not WPL.
    Increment and BASH VPOS.

$3648-366B -- SAVE TEXTFILE SETUP

    Zero save/adjust flag $3647. Use auxiliary memory as file
    source by setting $71. Save the old filename to the "="
    buffer. Print old filename to user prompt. Get any user
    changes to the filename. If a "?", then catalog disk and
    restore old filename, and try again. Abort on a carriage
    return. On any other filename, fall through to next module.

$366C-36C9 -- SAVE ENTIRE TEXTFILE TO DISK

    Set $AD flag to use old filename. Read the last character
    in  the filename. If "+" then set the Append flag $E2 and
    zero the "+" out of the filename. If not, clear $E2. Copy
    LOCURS to $98,99, remembering present cursor location when
    finished  saving. If adjust, bypass processing that follows.
    If save, filter  filename for special delimiters. If found,
    process via next module. Move cursor to end, putting
    everything  in LOFILE. Save file to disk. Update screen
    markers. Reset  old string flag and adjust flag. Move
    characters from LOFILE to HIFILE to restore old cursor.

$36CA-3726 -- SAVE PART OF TEXTFILE TO DISK.

    Process special delimiters. Zero last delimiter. Save the
    present  cursor position to auxiliary utility pointer. This
    transfers to the actual disk write routines as a starting
    address. Begin moving characters from HIFILE to LOFILE,
    one at a time.Compare each character against the first
    delimiter string character. If no match, keep scanning
    characters. If a match try to match next character in the
    delimiter string. Substitute carriage returns and bypass
    wildcards as needed. If no perfect match, abort by moving
    all the characters back to the original LOCORS position, and
    then restoring the old filename. If match found, write to
    disk, using $AE,AF to mark the start of save stuff, and
    $84,85 LOCURS to mark the end. Then update the screen
    markers, reset the string flag $AD, move everything back to
    the original LOCURS position, and restore the old filename.
```

Listing C.9-cont...

```
$3727-3747 -- SET FILENAME COUNT

    Move the filename at $8700 into the filename hold at $1F00,
    carefully shifting everything one to the right to make room
    for the length count. Count only numerals, letters, commas,
    and  periods, stopping on any control command or any other
    punctuation. Save the length count to $1FO0.

$3784-377C -- WRITE TEXTFILE TO DISK SETUP

    Force entire file to low ASCII. Abort if an adjust, rather
    than a save. Find length by subtracting LOCURS from the aux
    utility pointer $AE, AF. Open filename. If append, set mark
    via MLI. If not append, set end of file via MLI. Save as
    many sectors  to disk with next module. Close file.

$377D-37DC -- WRITE ONE SECTOR TO DISK

    Read first character of the portion of the text file to be
    saved into $B900. Continue copying until no more characters
    needed  or until buffer fills at 512 characters. On each
    buffer fill, write  a new disk sector. Read aux or main
    memory per flag $71, using aux memory for normal save. Save
    length  to data buffer. Exit via ProDOS error processor.

$37DD-37F8 -- CLOSE FILES

    Enter at $37DD to close all files. Enter at $37E5 to close
    one file. Poke file reference number to MLI buffer, using
    $00  for all files, and the number on any one file. Close
    files with  a ProDOS Close MLI. Reset buffer to close all.

$37F9-380B -- ProDOS GET EOF MLI

    Find the length of the currently open file, reporting to the
    buffer at $380C. Load the registers with the possible 24 bit
    length result, with X = MSB (normally zero) Y = middle 8
    bits, and A= LSB.

$3811-3830 -- ProDOS SET MARK MLI

    Copy the reference number from set mark to get EOF. Save
    new end-of-file marker from X register MSB (usually zero),
    Y register middle eight bits, and Accumulator LSB. Set mark
    MLI, for the new endpoint to the file. Used for Append.

$3831-387C -- POSTFIX SLOT AND DRIVE SETUP

    Abort if the first character in the filename is not a comma
    or period. Zero the slot and drive stashes at $38D3, 38D4.
    Read the filename. Force upper case. If a slash is found,
    hold distance to slash in $38D4, and fall through to next
```

Listing C.9-cont...

```
    module. If nothing beyond comma or period, also fall through
    to next module. If  D for drive, insert 0 for drive 1 or 1
    for drive 2 into $38D3 MSB. If a .s for slot, force range to
    0-7  and shift to put $38D3 into ProDOS DSSS 0000 format.

$387D-38D5 -- POSTFIX SLOT AND DRIVE

    Check the DSSS 0000 stash. If slot zero, force slot six
    instead. Get volume name from Prooos on-line MLI using this
    slot and drive. Truncate to sixteen characters max, loading
    into $B900 pathname buffer. If a postfix exists, append it
    onto the path name. Add a slash to the start of the pathname,
    and Then move the pathename to the $1FOO buffer. Count the
    characters in the pathname and save to $1FOO.

$38D6-38ES -- FIND PATHNAME OF INTENDED SLOT AND DRIVE

    Store the DSSS 0000 in the accumulator into the MLI buffer.
    Do ProDOS On-line MLI to load name of disk in target drive
    into path name buffer at $B900. Exit to error processor.

$38E6-38FD -- OPEN TEXT FILE

    Calculate filename length and save to $1FOO. Use $04 text
    file  and reference number $01.Open file via ProDOS Open
    MLI. Transfer reference number other MLI links . Get EOF
    of current file and save to EOF stash at $3918 (low), $3919
    (med), and $391A (high).

$391B-3928 -- ProDOS SET EOF MLI

    Reads the current open file and appends the current end
    position.  Position must have been pre-poked into $3926
    (low), $3927 (med), and $3928 (high). Exit to error proc.

$3929-3949 -- PROCESS SPECIAL DELIMITERS

    If the usual "/" delimiter that does not allow any fancy
    stuff,  put an $01 into $EC, the any length stash, an $02
    into $EB the fake carriage return stash, and an $03 into
    the $EA wildcard stash. If a special delimiter is used, put
    the next higher ASCII character into $EC, the next one
    after that into $EB, and the next one after that into $EA.
    For instance, a "<" delimiter will have an any length "=",
    a fake carriage return ">", and a wildcard "?". Note that
    four ASCII characters follow each other in sequential order.

$394A-398D -- LOAD SETUP

    Hold current LOCURS position in $BA, BB formatting pointer.
    Set the old string flag $AD to $FF. Set the memory load
```

Listing C.9-cont...

```
flag $71 to $FF for a load into auxiliary memory. Save old
filename to the "=" file. Force normal case and  load rather
than append,  and included delimiters. Print filename to user
prompt  and get any changes, saving new filename length to
$E9 stash. Read the last filename character and compare it
to UT, usually a "" for screen-only load. If a screen only
load, zero the symbol and set the screen-only flag $FD. If
a question mark as the first character, do a catalog and try
again. If a carriage return, fall thru to cleanup. If a legal
filename, fall thru tothe  next module.

$398E-3A0E -- LOAD PROCESSING

Zero out the three delimiter stashes at $A2-A4. If the first
character is a "#" for copy from memory, then set the copy
from memory flag $77. Scan the filename for delimiters. If
If found, then process special delimiters, and zero the final
delimiter out of the filename. If delimiters exist, scan the
filename and save positions to $A2-A4. If delimiters exist,
check the final character for "A" or "N". If "A", set all
occurance flag $D2. If "N", set exclude delimiters flag.
Set up disk or  memory read. If no delimiters, read directly
to textfile from textfile or disk. If delimiters, fall thru
to next module.

$3A0F-3AAB -- LOAD WITH DELIMITERS

Get first character from $B900 buffer and hold to the line
justify buffer. Read load string. If a wildcard, get next
character. If a fake carriage return, substitute a real
carriage return.  Compare for a match. If no match, get
next character. If a match and if delimiters to be included,
then continue matching and put first delimiter into memory.
Continue reading characters, watching for second  delimiter
 if present. If third delimiter is present and delimiters
are to be omitted, swallow end delimiter string from textfile
to swallow buffer. If no third delimiter, read characters
till end of file. If all occurance flag is set, repeat search
for new first delimiter string as often as needed. Fall
through to load cleanup module below.

$3AAB-3ACA -- LOAD CLEANUP

Close file. If load to screen only, put down return prompt.
Restore old filename if it exists in the "=" file. Reformat
screen margins.
```