INTRODUCTION
HOW TO...
SHELLING COMPILED PROGRAMS

TABLE OF CONTENTS

YOU NAME & ADDRESS

Beagle Bros Product Registration Card

THE BEAGLE COMPILER

Applesoft Speed-Up Program

by Alan Bird

Published by Beagle Bros Micro Software, Inc.
San Diego, California 92110

WELCOME TO THE BEAGLE COMPILER!
The Beagle Compiler does one thing—it rewrites Applesoft BASIC programs so they run faster, just as if they were written in machine language. Machine language programs run much faster than Applesoft programs because no time is wasted interpreting 'human' commands like HOME, GOTO, IF, THEN, and so on. While the Beagle Compiler doesn't actually convert programs into machine language (it actually converts them into its own language), the effect is the same.

Applesoft BASIC may be slower than machine language, but it is far easier for most of us to write programs with—and it's easier to read. The Beagle Compiler gives you the best of both worlds—easy-to-write programs and machine language speed.

BACK IT UP
The Beagle Compiler disk is not copy-protected, so you can (and should) make a backup in case something happens to the original. Use the copy program that came with your Apple, or the 35-second DISK.COPY program from our Extra K disk. You may also transfer files from disk to disk using our Big U disk’s FILEMOVER program or one of Apple's utilities.

Please don't give copies of our disks and programs away to your friends. Every illegal copy is a vote for copy protection and against friendly software. If you plan on giving copies of your own compiled programs away, read page 6.
COMPILER FACTS
Just like it says in the ads, "after you boot the Beagle Compiler disk, you can run almost any Applesoft program at machine language speed. FAST!"

HOW FAST IS IT?
Unscientific testing shows that compiled programs tend to run between 2 and 15 times as fast as Applesoft programs; it depends on what the program actually does. Some functions like string and variable manipulations show a tremendous speed increase. Other things like floating point calculations aren't affected at all.

WHEN SHOULDN'T YOU COMPILe?
Machine language speed isn't always an advantage; some programs, like question-and-answer programs, work just fine in plain old Applesoft. Too much speed will make many programs impossible to use.

Some programs that benefit from compiling may have certain sections that will need to be slowed down. Since you can't compile just part of a program, you'll have to make adjustments in Applesoft before compiling.

WHEN CAN'T YOU COMPILe?
Most Applesoft programs compile with ease. Occasionally, a program will be too large to compile or contain commands that are incompatible with the compiling process.

Non-Applesoft programs won't compile. (You can't compile AppleWorks for example; it's not written in Applesoft).

Copy-protected programs won't compile unless you unprotect them first. Don't ask us how—we don't know how.

DOS 3.3 programs will usually compile after you convert them into ProDOS (use one of Apple's programs to do the converting). Make sure a converted program works before you compile it.

SPECIAL BENEFITS OF THE BEAGLE COMPILER
The Beagle Compiler is better than any compiler we've ever seen. And we've seen a few.

- Other compilers do not support ProDOS.
- Other compilers will not compile programs "on the spot" using the standard RUN command.
- Other compilers produce code that is significantly larger than the original program. The Beagle Compiler does the opposite.
- Other compilers take minutes instead of seconds to convert programs. And then you still might have problems.
- Other compilers choke on common Applesoft statements like HMEM, LOMEM, DEFFN, etc.
- Other compilers require many more program changes than the Beagle Compiler requires. For example:

10 MAX=1000: DTM A8 (MAX), B5 (MAX), C5 (MAX*2)
Other compilers would look askance at the above program line and make you change it to:

10 MAX=1000: DTM A8 (100), B5 (100), C5 (200)
Then you have to recompile.

THERE ARE SOME MEMORY RESTRICTIONS
Booting the Beagle Compiler disk will cost you about 11K of main memory. You can cut this figure in half by loading only one of the Compiler's two files—see page 26.

ABOUT PROGRAM EDITORS
Most Applesoft programmers use some kind of program editor. Unfortunately, you cannot have non-relocatable programs like Beagle Bros G.P.L.E. (Global program Line Editor) in memory with the COMPILER.SYS file—sorry, there just isn't room.

However PROGRAM WRITER by Alan Bird (!) will work just fine.

If you are hooked on G.P.L.E., boot normal ProDOS to use G.P.L.E. to write and test programs, then boot the Beagle Compiler to run them at compiled speed. The COMPILER file (see page 26) can be in memory with G.P.L.E. as long as you install G.P.L.E. first, then the COMPILER file.

To use PROGRAM WRITER, you must install things in the proper order: (1) the COMPILER.SYS file, (2) PROGRAM WRITER-language card version, (3) the COMPILER file.
SELLING (OR GIVING AWAY)
COMPiled PROGRAMS

You may legally sell or give away copies of programs that you own and have compiled with the Beagle Compiler. Since the Compiler itself is protected by copyright laws, the recipient of your programs must use his or her own purchased copy of the Compiler to run them.

There is an alternative: If you want to include the Beagle Compiler's COMPILER SYSTEM file on disks that you will be selling or giving away, you may do so after paying a very reasonable licensing fee to the Compiler's author, Alan Bird. Call or write for more information:

The Software Touch
C/O Compiler Licensing
9625 Black Mountain Road, #204
San Diego, California 92126

Or phone The Software Touch: (619) 549-3091

After a licensing contract has been signed and fees paid, only the file COMPILER SYSTEM may be put on the disk you are selling or giving away. This is the file that actually runs compiled programs. Under no circumstances are you permitted to include the COMPILER file on disks that you sell or give away.

IS THIS MANUAL UP TO DATE?
RUN NOTES NOW TO FIND OUT.

Run the Applesoft NOTES program on the Beagle Compiler disk to learn about any changes or corrections that apply to this instruction manual.

THE BASiC FACTS
In writing this manual, we assume you know the "basics" about loading and saving files, running Applesoft programs and so on. Even if you don't, you still should be able to reap most of the benefits of the Beagle Compiler by reading pages 1-15.

HOW TO USE
THE BEAGLE COMPILER

RULE #1:
APPLESOFT PROGRAMS ONLY

The Beagle Compiler only works with unprotected ProDOS-based
Applesoft BASIC programs. You must have a copy of the program
you want compiled saved on a ProDOS disk.

When you catalog a disk (by typing CAT), "BAS" identifies a
file as being Applesoft BASIC:

/SAMPLE.DISK
NAME TYPE
FILE.A BAS "You can compile BASIC programs.
FILE.B BIN "You can't compile binary files.
FILE.C TXT "You can't compile text files.
FILE.D SYS "Forget it.
FILE.E VAR "Ditto.
FILE.F COM* "There is a program that has been
saved in compiled format.

*COM will read as "INT" if the Compiler isn't in memory.

RULE #2:
BE SURE THE COMPILER IS INSTALLED

The Compiler's commands won't work until you "install" the
Compiler in your Apple's memory. The easiest way to do this is
to BOOT THE BEAGLE COMPILER DISK (put the disk in your
main drive and turn on your Apple).

There are other ways to install the Compiler that save
memory and/or disk space—see pages 26 and 27.

RULE #3:
WATCH OUT FOR CERTAIN THINGS

- ProDOS's CHAIN, STORE and RESTORE commands make
  programs require special treatment—see pages 18 and 19.

- Ampersand (&) statements with parameters and the routines
  that they call must be altered by someone with assembly
  language experience—see page 20.

- Some Applesoft commands are not compilable—specifically
  CONT, DEL, LIST, LOAD, NOTRACE, RECALL, SAVE, SHLOAD,
  TRACE and STORE (ProDOS's LOAD, SAVE and STORE will
  compile). Removing any of these commands will not harm
  99.6502% of the programs we have seen.

- Weird memory pokes are unpredictable. If the program you
  want to compile pokes values into Zero Page or BASIC.SYSTEM
  or some other exotic place, go ahead and try compiling—if
  you're lucky, you'll have no problems at all.

- Giant Applesoft programs are usually compilable if you
  compile them to disk without the COMPILER.SYSTEM file in
  memory—see page 27.

EXPERIMENT
WITH TESTPROGRAM

There is a short
Applesoft program
called TESTPROGRAM
on the Beagle
Compiler disk. No
big deal, but it uses a
lot of Applesoft
commands and
serves as a good
demo of how the
Compiler works. In
the examples on the
following pages,
almost any
Applesoft program
may be substituted
for TESTPROGRAM.
HOW TO RUN AN APPLESOFT PROGRAM AT COMPILED SPEED

This is easy. First, be sure the Compiler is installed in memory (it is if you have booted the Beagle Compiler disk). Now just run your program like you always do, by typing:

RUN NAME
or -NAME

(NAME is the name-or pathname-of your Applesoft program.)

After a brief "COMPILING..." message, your program should be running at machine language speed. If this isn't the case, the Compiler probably isn't installed. If you see an error message on the screen, read pages 20-22.

Since compiled programs have no line numbers, the PRODOS command RUN NAME, @123 will not work (123 represents any program line number).

QUITTING A PROGRAM

You can often quit an Applesoft program by pressing Control-C. This might not work, however, when you're running at compiled speed (see page 12 for a quick fix).

Control-Reset will almost always let you quit. Some programs, however, are written so you can't quit no matter what, and you may need to reboot to escape.

RE-RUNNING A PROGRAM

After you quit running a compiled program, you may type RUN to re-run it. If you get a NOT A COMPILED PROGRAM error message, something has disturbed the compiled program in memory.

EXAMPLE

To run TESTPROGRAM from the Beagle Compiler disk at compiled speed:

1. Boot the Beagle Compiler disk.
2. Type RUN TESTPROGRAM
or type -TESTPROGRAM
3. To stop the program, press Control-Reset.
4. Type RUN to run TESTPROGRAM again.

HOW TO SAVE A PROGRAM ON DISK IN COMPILED FORMAT

To save a compiled version of an Applesoft program on the current disk, type:

COMPILE NAME, NEWNAME

(NAME is the name-or pathname-of your Applesoft program on disk. NEWNAME is the name-or pathname-for saving the compiled file.)

Your Applesoft program will be loaded, compiled and then saved on disk. A FILE TYPE MISMATCH error message here might mean that you used the same name for both files. You may use the same name for the compiled file if you are saving onto another disk or directory. For example, you could type:

COMPILE /DISK1/NAME, /DISK2/NEWNAME

This command would load NAME from DISK1 and save it as NAME on DISK2 in compiled format.

Cataloging the disk will reveal compiled programs as type COM (that's COM for Compiled instead of BAS for BASIC). If you catalog without the Compiler installed, COM will appear as "INT".

EXAMPLE

To save TESTPROGRAM on disk in compiled format:

1. Be sure the Compiler is installed.
2. Insert the Beagle Compiler disk in drive 1.
3. Set the prefix if necessary by typing PREFIX, D1
4. Type COMPILE TESTPROGRAM, TESTFAST
   This will save a new version of TESTPROGRAM called TESTFAST on the disk. When you catalog the disk (by typing CATALOG), you will see TESTFAST listed as a COM file.

WHY SAVE IN COMPILED FORMAT?

- You save time by not having to wait for compiling each time you run the program.
- You save disk space because compiled files are generally smaller than Applesoft files.
- You save memory space because the COMPILED file (see page 26) doesn't need to be in memory when you run the program.
- Your programs aren't listable and snoopers can't look at them and change them. (This can be a disadvantage).
HOW TO RUN A COMPILED PROGRAM FROM DISK

You run a compiled (COM) file from disk the same way you run an Applesoft (RAM) file. Type:

RUN NAME
or -NAME

(NAME is the name—or pathname—of your compiled program.)

EXAMPLE
To run TESTPROGRAM in compiled format:
1. Compile TESTPROGRAM so it creates the COM file TESTFAST (follow the steps in the previous example).
2. Type RUN TESTFAST or -TESTFAST

The COMPILER.SYSTEM file must be installed to run compiled programs (it is if you booted the Beagle Compiler disk). You can save memory by not installing the COMPILER file—see page 27.

HOW TO MAKE A CHANGE TO A COMPILED PROGRAM

You can't change a compiled program. Instead, change the Applesoft "source" program (the one you compiled in the first place). Make changes the way you always do, and always be sure to save the changed program to disk before recompiling.

Read the note about program editors G,F,L,E, and PROGRAM WRITER on page 5.

EXAMPLE
To make a change to TESTPROGRAM:
1. Type LOAD TESTPROGRAM
2. Type LIST 10 to see program line 10.
3. Type 10 X=5 to change line 10. This will have the effect of changing the patterns on the screen when the program is running.
4. Type SAVE TEST2 to save the Applesoft change.
(Any legal file name may be used.)
5. With the Compiler installed, type RUN TEST2 or -TEST2 to run at compiled speed.
   Or type COMPILE TEST2, NEWTEST2 to save in compiled format.
6. To make more changes type LOAD TEST2 and go back to step 2.
HOW TO MAKE CONTROL-C STOP A COMPILED PROGRAM

Normally Control-C will stop an Applesoft program but have no effect on compiled programs (except in response to INPUT statements). To make Control-C halt a compiled program, add a RESUME statement somewhere in the program you are going to compile. Think twice before using this technique, because it will have the side effect of making your compiled program run somewhat slower.

The RESUME statement has an undesirable effect on Applesoft programs, so you should put it somewhere where it can’t possibly get executed—like after the end of your program (END: RESUME).

EXAMPLE
To allow TESTPROGRAM to be halted with Control-C after it is compiled:
1. Type LOAD TESTPROGRAM
2. Type 60000 END: RESUME to add program line 60000.
   (60000 may be replaced with any line number 1-63999; just be sure the RESUME statement doesn’t get executed.)
3. Type SAVE TEST2 to save the changed version.
   (Any legal file name may be used.)
4. Type TEST2 to run the program.
5. Press Control-C to stop the program.

A NEW BREAK MESSAGE
When your compiled program is stopped by Control-C (or a STOP statement or an error), you will see an error message something like BREAK AT 50ABC. This tells you the hexadecimal address in memory where the program stopped (instead of which line number, because there are no line numbers in compiled programs). See page 23 for more information about this number.

HOW TO RUN A PROGRAM AT NORMAL SPEED

The best way to run a program at normal speed is to remove the Compiler from memory by booting a normal ProDOS disk.

Warning:
The method described below is not guaranteed to work. In fact, with certain programs, it could cause serious problems that require you to reboot.

Always save your programs before running them!

After taking the warning above into consideration, load an Applesoft program from disk and type :RUN. Notice that this command begins with a colon (:). If you omit the colon you’ll get a NOT A COMPILED PROGRAM error message.

EXAMPLE
To run TESTPROGRAM at normal speed with the Compiler installed:
0. Read the Warning above.
1. Type LOAD TESTPROGRAM
2. Type :RUN
3. To quit, press Control-C or Control-Reset.
4. To see it again, type :RUN again
HOW TO SPECIFY A NEW ADDRESS
FOR A PROGRAM
(for advanced programmers)

Compiled programs normally load and run at address 2049 (S0801), just like Applesoft programs. You may run a compiled program at a different address by changing the Applesoft program before it is compiled.

For example, insert the following line at the beginning of TESTPROGRAM to run it above hi-res page 1 at 16384 ($4000):

```
1 IF PEEK(104)<>64 THEN POKE 16386,0: POKE 104,64: PRINT CHR$(4)"RUN TESTPROGRAM"
```

Replace the 64's with 96's and the 16384 with 24576 to load the program above hi-res page 2 at 24576 ($6000).

You may also specify an address with a RUN command followed by a comma and the address. The following command will compile and run an Applesoft program--run a compiled program--above page 1 and page 2 respectively:

```
RUN NAME, A$4000
RUN NAME, A$6000
```

(Note: In this procedure, "RUN" cannot be replaced with a hyphen.)

HOW TO WRITE PROGRAMS
FOR MAXIMUM COMPILED SPEED
(for advanced programmers)

The one major point to keep in mind to make compiled programs run faster is to avoid using floating point values whenever possible. The Compiler does all of its math using integer values whenever it can--integers process much faster than floating point values.

Floating point is used:
- when a value has a fractional part (i.e. 3.5).
- when a value is greater than 32767 or less than -32767.
- when division is used in an expression.
- when any of the following functions are used:
  - ATN, COS, EXP, LOG, RND, SIN, SQR or TAN

BASIC TECHNIQUES DON'T APPLY
The following programming methods DO speed up Applesoft programs but they DO NOT speed up compiled programs (they also don't do any harm):
- Using real variables instead of integer variables.
  (Using A%.3 instead of A=3 will not affect a compiled program's speed.)
- Using variables instead of numeric constants.
  (In a compiled program, A=PI executes no faster than A=3.14159.)
- Putting frequently-executed lines and subroutines near the beginning of a program.
- Putting frequently-used variables near the beginning of a program.
HOW TO COMPILe PROGRAMS THAT USE
THE CHAIN COMMAND

The ProDOS CHAIN command works just like the ProDOS RUN
command, but existing variables stay intact.

Programs that use CHAIN share common variables and
must be given special treatment for compiling to be successful.
Otherwise a FILE TYPE MISMATCH error will occur.

All programs involved with a CHAIN command must be
compiled to disk using the COMPILe command to compile one
program and a special COMMON command to compile the other
program(s). COMMON's syntax is similar to COMPILe:

COMMON NAME, NEWNAME

COMMON must be used immediately after COMPILe. If you later
make a program change or add a new file that will CHAIN to or
from the existing (already compiled) files, you must start over
and recompile all of the files.

EXAMPLE

Say you have three programs—MAIN.PROG, PROG.A and
PROG.B—thata share variables. MAIN.PROG is the startup
program and it will CHAIN to PROG.A which will CHAIN to
PROG.B which will CHAIN back to MAIN.PROG. Here's what
you do to compile these programs:

1. Compile MAIN.PROG with the usual COMPILe command:
   COMPILe MAIN.PROG, MAIN.COMP
   (Any legal file name may be used.)

2. Immediately compile each program that is to share data
   by using the COMMON command:
   COMMON PROG.A, PROG.A.COMP
   COMMON PROG.B, PROG.B.COMP

3. To run the program(s) with the Compiler installed, type:
   RUN MAIN.COMP
   or -MAIN.COMP
   (Note: Even the COMMONed files may be run.)

HOW TO COMPILe PROGRAMS THAT USE
STORE AND RESTORE COMMANDS

Note: This page applies to the ProDOS STORE and RESTORE
commands. The Applesoft RESTORE command will compile just
fine. The Applesoft STORE command is obsolete.

The ProDOS STORE command normally saves the variables in
memory on disk in a VAR file. RESTORE loads these variables
back into memory.

Programs that use STORE and RESTORE share common
variables and must be given special treatment for compiling to
be successful. The programs must be compiled to disk using the
COMPILe command on one program and the COMMON command
on the other (see CHAIN, previous page).

COMMON must be used immediately after COMPILe. If you
later make a program change or add a new file that shares
variables with the existing (already compiled) files, you must
start over and recompile all of the files.

STORE and RESTORE in compiled programs create and use
variable files of type CVR instead of VAR.

EXAMPLE

Say you have a program called DATA.SETUP that uses the
STORE command to write variables that will be loaded
(using RESTORE) by a program called BIG.GAME. Here's what
you do to compile these two programs:

1. Type COMPILe BIG.GAME, BIG.GAME.COMP
   (Any legal file name may be used.)
2. Immediately type:
   COMMON DATA.SETUP, DATA.SETUP.COMP
3. To run either program with the Compiler installed, type:
   RUN BIG.GAME.COMP or -DATA.SETUP.COMP
HOW TO WRITE AMPERSAND ROUTINES
FOR COMPILED PROGRAMS
(For advanced assembly language programmers only.)

An ampersand routine without parameters (& by itself) will compile just fine. Ampersand routines with parameters (like &XXX or &XXX,YYY,ZZZ) are a different story. Both the ampersand command and the machine language routine itself must be modified.

CHANGE #1:
USE & & INSTEAD OF &
When calling an ampersand routine from a compiled program, you must use two consecutive ampersands (for example, you would use & & SORT instead of & SORT). This is how the Compiler detects programs that have or have not been modified.

CHANGE #2:
RE-EVALUATE YOUR PARAMETERS
With the Compiler installed, a JSR to $98FD will evaluate the next parameter after an ampersand:
- If the parameter is a string, a pointer to the string will be found at $5F, $5E. All strings in a compiled program are stored with the length in the first byte.
- If the parameter is a numeric value and the carry flag is clear, the value is an integer with its low byte in the X-register and the high byte in the Accumulator.
- If the parameter is a numeric value and the carry flag is set, the value is floating point and stored in the FAC (at $9D).

There is no way for the Compiler to determine if the correct parameters are being passed to your ampersand routines. If the correct parameters are not there, the program will most likely crash miserably.

EXAMPLE
Let's write an ampersand routine that prints the first character in string S, N times. Our Applesoft program looks like this:

```
20  SS="CO8": N=80
50  & & N, SS
```

This program's mission is to print 80 Cs. The assembly code would look like this:

```
JSS  $98FD ; evaluate N
BCS ERROR ; reject floating pt.values
CMP #0 ; evaluate high-byte
BNE ERROR ; don't accept anything>255
PHA ; save N
JSR $98FD ; evaluate SS
PLA ; restore N
TXA ; use as counter
BEQ DONE ; counter was 0
LDA (SF6),Y ; get length of string
BEQ DONE ; null string
INY
LDA (SF6),Y ; get 1st character in string
ORA $80 ; set MSB
LOOP JSR $98FD ; print character
DEX
BNE LOOP ; loop
DONE RTS ; return to compiler
ERROR LDX #53 ; "ILLEGAL QUANTITY ERROR"
JMP $98FD ; handle error
```
COMPiled PROGRAM EXECUTION ERRORS

An Applesoft or ProDOS program error will cause a compiled program to crash just like an uncompiled program. The only difference is that compiled programs produce strange error messages like:

?ILLEGAL QUANTITY ERROR AT 50ABC.

Uncompiled programs, as you know, produce messages like:

?ILLEGAL QUANTITY ERROR IN 123.

In this comparison, 50ABC is the hexadecimal location (address) in memory of the error, and 123 is the line number of the error (compiled programs have no line numbers).

Since line numbers are easier to work with than memory locations, the most efficient way to trap errors is to test your programs and get them working correctly before you compile.

THE PRINT.LINES PROGRAM CONVERTS $ADDRESSES TO LINE NUMBERS
(For advanced programmers)

If you insist on ignoring our advice above: To determine the line number that is equivalent to a hex error address, compile a program using the COMPile or RUN command, then:

1. (optional) Turn on your printer by typing PR#1.
2. With the Beagle Compiler disk in the current drive, type: BRUN PRINT_LINES or -PRINT_LINES
3. Type PR#0 to deactivate your printer if necessary.

The numbers produced by the PRINT_LINES program are the starting hex addresses and matching decimal line numbers.
ERRORS FOUND DURING COMPILING

If your Applesoft program contains errors, the Compiler will do its best to find them during the compiling process. Each offending line will be listed with the approximate location marked. The Compiler will then ask "CONTINUE WITH ERRORS?" to see if you want to go ahead and run the program anyway (some errors are intentional or cause no problem). If you answer no, compiling will stop so you can make repairs. Save your repaired program on disk; then recompile it.

Many errors will not be found by the Compiler; this includes most Prodos errors and errors inside quote marks. Here are some common errors that will be found during compilling:

<A> (SYNTAX ERROR)
• A <A> symbol could mean your program contains an Applesoft keyword that is unacceptable to the Compiler—specifically CONT, DEL, LIST, LOAD, NOTRACE, RECALL, SAVE, SHLOAD, STORE and TRACING. Programs with these commands will not compile. (Note: Prodos's LOAD, SAVE and STORE will compile).
• Other culprits are those you have encountered before, such as missing parameters (like HIPLOT with no coordinates), type mismatches (like AS=5), misspelled keywords (like PIRNT), missing commas and colons, and so on. Your Applesoft instruction manuals will help you make program repairs.

<#> (UNDEFINED LINE NUMBER ERROR)
A <#> symbol usually means your program used a GOTO or COSUB to a nonexistent line number.

<A> (ARRAY DIMENSION ERROR)
An <A> means your program has illegally allocated arrays. For example: A(25)=3; A(6,3)=3

<*> (ILLEGAL QUANTITY ERROR)
The only illegal quantities the Compiler will find are illegal addresses (for example, POKE 90000,0). Other illegal quantities (like HIPLOT 90000,0) won't be noticed until your program crashes.

INCLUDE APPENDED MACHINE CODE?
This message means the Compiler has found some extra space at the end of your program. This could be useless garbage or it could be valuable data or a routine that is called by the program. When unsure, play it safe by answering Y (Yes, include the code).

KEYBOARD ERRORS

These errors may occur immediately after you type a command:

FILE TYPE MISMATCH
• Maybe you used the same two names when compiling a file to disk (COMPILE NAME, NAME).
• Or you used a command like COMPILE NAME, NEWNAME and NEWNAME was already on the disk as a type other than COM.
• Or you ran a program that uses CHAIN, STORE or RESTORE and you didn't compile with the COMMON command (page 18-19).
• Or you wrote an Applesoft (BASI) STARTUP program. STARTUP must be a compiled (COM) file.

NO BUFFERS AVAILABLE
• You may have tried to install the the COMPILER.SYSTEM or COMPILER file more than once. One time is enough.
• Or you may have tried to run a program below address 50801.
• Or you may have pressed Control-Reset during a catalog.
Solution: Try again or reboot.

NOT A COMPILED PROGRAM
With the Compiler in memory, you typed RUN after loading an Applesoft program. See pages 10 and 15.

PATH NOT FOUND
Translation: File Not Found. If you're sure the not-found file is on the disk and you spelled its name right, try typing PREFIX/ or PREFIX, S6, D1 (use your slot and drive numbers) or PREFIX/DIR/SUB (use your directory/subdirectory names).

PROGRAM TOO LARGE
• Your Applesoft program is too large to fit in memory.
• Or you are trying to run a program at too high an address.
Try compiling to disk, then removing the COMPILER file (page 27).

SYNTAX ERROR
• Maybe you spelled a command wrong.
• Or maybe you used the COMPILE or COMMON command without installing the Compiler.
• If you get a SYNTAX ERROR as a response to typing something you know is legal (like "LIST"), memory is probably damaged, and you should reboot. Try pressing Control-Reset first. (It might not help, but it feels kind of good.)
COMPILER AND
COMPILER.SYSTEM

When you catalog the Beagle Compiler disk, you will see the two files COMPILER (BIN) and COMPILER.SYSTEM (SYS) listed.

- COMPILER is the program that converts Applesoft programs into compiled format.
- COMPILER.SYSTEM is the program that runs programs that have already been compiled.

When you boot the Beagle Compiler disk, here's what happens:
1. COMPILER.SYSTEM is installed into memory.
2. STARTUP is loaded and run.
3. COMPILER is installed in memory.

If no COM file named STARTUP exists on the disk, the above process stops after step 1. Beagle Compiler's STARTUP loads COMPILER, although you may change that if you like by replacing STARTUP with your own version. STARTUP must be a COM file.

OMIT COMPILER TO SAVE SPACE

If you are only going to be running compiled (COM) programs and not converting Applesoft (BAS) programs, you can conserve about 6K of memory by not installing the COMPILER file. Any one of these methods will do the trick:

- Rename the STARTUP file on the Beagle Compiler disk before booting. Then boot the disk.
- Or, replace Beagle Compiler's STARTUP with your own version that doesn't install COMPILER. STARTUP must be a COM file.
- Or, copy the COMPILER.SYSTEM file onto another disk that contains the file PRODOS (but not COMPILER or BASIC.SYSTEM). Boot this disk and you will be able to run compiled programs, but not convert Applesoft programs.

OMIT COMPILER.SYSTEM TO COMPILE LARGE PROGRAMS

If you are going to compile a very large Applesoft program, there may not be room in memory for your program and both Beagle Compiler files. A solution might be to install the COMPILER file without COMPILER.SYSTEM, then compile your Applesoft program to disk.

To prevent COMPILER.SYSTEM from loading, boot a normal ProDOS disk that loads BASIC.SYSTEM, then insert the Beagle Compiler disk and type the command BRUN COMPILER (or -COMPILER). Do this only once because COMPILER eats 6K of memory each time it's installed.

Remember, COMPILER.SYSTEM will have to be installed (alone or with COMPILER) to actually run compiled programs.
**MENU**

**MENU** is a COM file that lets you select disk drives and programs from an AppleWorks-style menu. You can make **MENU** run when you boot a disk by renaming it **STARTUP**. Or, you can make your **STARTUP** program run **MENU**.

To get **MENU** going, type `-MENU`. A list of all of the available ProDOS drives will appear at the top of the screen. "S6,D1" means Slot-6 Drive-1, "S6,D2" means Slot-6 Drive-2, etc. S3,D2 represents ProDOS's RAM disk. Below that will be all of the executable files (BAS, BIN and COM) on the highlighted drive.

Do this to run a program from one of your drives:

1. **Press the "< >" keys or a number** to highlight the drive number that contains the program you would like to run.
2. **Press the ARROW keys** and/or the **TAB key** to highlight the program you would like to run.
3. **Press the RETURN key** to run the highlighted program.

If a subdirectory is highlighted when you press **RETURN**, its file names will be displayed—go back to step 2.

To quit **MENU** at any time, press the **ESC** key.

**MENU ERROR MESSAGES**

- **I/O ERROR** might mean a drive door is open.
- **NO DEVICE ERROR** usually means you are trying to read a slot's drive 2 when no drive 2 exists.
- **PATH NOT FOUND** probably means you switched disks.

**OTHER POSSIBLE PROBLEMS**

- **If a program crashes**, it probably wasn't written to be run (for example, it might be a hi-res picture instead of a program).
- **If you don't see a program listed on the screen** and you know it's on the disk, it might be a non-executable file type like TXT or VAR. Or there might not be room for it on the screen. The limit in 80-columns is 60 file names/10 disk drives. In 40-columns the limit is 30 file names/5 drives.

**ENHANCEMENTS TO COMPILER.SYSTEM**

The programs on this page make patches to the Compiler. Just **RUN** the file after booting the Beagle Compiler disk (**COMPILER.SYSTEM** must already be in memory).

**INPUT.ANYTHING**

This patch replaces the Compiler's **INPUT** statement with one that allows commas and colons. This is very handy when inputting data from text files.

**SLOW.PDL**

Installing **SLOW.PDL** puts a small delay in the PDL (paddle) function so you will always get the correct reading.

The Applesoft BASIC Programmer's **Reference Manual** recommends that if you are doing consecutive reads of the game paddle or joystick with the PDL function, that you put a small loop in between the reads such as **FOR X = 1 TO 10**. Because the Compiler causes programs to run much faster, you will have to increase these delays. Or utilize **SLOW.PDL**.

**FAST.HPLOT**

**FAST.HPLOT** replaces the **HPLOT** statement with one that is much faster. It's not normally installed in the Compiler because it takes up a considerable amount of memory. Don't use this patch unless your programs are heavily into hi-res plotting.

```
1 CALL-3109: POKE 230,32: CALL-3086: HCOLOR=7: HPLOT
9,9: CHERR GOTO 1
2 HPLOT TO RND(1)*290, RND(1)*200: RUN 2
```

**WOULD YOU JUST HGR IT FOR A WHILE?**
### APPLESOF/PRODOS COMMAND SUMMARY

Use this list for reference. For more complete information, check the nearest Applesof or ProDOS instruction manual.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A ABS(x)</td>
<td>Absolute (positive) value of x</td>
</tr>
<tr>
<td>A AND</td>
<td>Logical &quot;and&quot; in an IF statement</td>
</tr>
<tr>
<td>P APPEND f</td>
<td>Add data to a sequential text file</td>
</tr>
<tr>
<td>A ASCII(&quot;A&quot;)</td>
<td>ASCII value of a character</td>
</tr>
<tr>
<td>A ASCII(A$)</td>
<td>ASCII value of a string's first character</td>
</tr>
<tr>
<td>A AT</td>
<td>See DRAW, XDRAW, HLIN and VLIN</td>
</tr>
<tr>
<td>A ATN</td>
<td>Arctangent of x in radians</td>
</tr>
<tr>
<td>P BLOAD f</td>
<td>Load binary file f</td>
</tr>
<tr>
<td>P BRUN f</td>
<td>Load and execute binary program f</td>
</tr>
<tr>
<td>P BS nelln</td>
<td>Save data; Address n, Length m</td>
</tr>
<tr>
<td>A CALL n</td>
<td>Branch to machine language routine at n</td>
</tr>
<tr>
<td>P CAT</td>
<td>Display disk contents in 40 columns</td>
</tr>
<tr>
<td>P CATALOG</td>
<td>Display disk contents in 80 columns</td>
</tr>
<tr>
<td>P CHAIN f</td>
<td>Run file f without clearing variables</td>
</tr>
<tr>
<td>A CHR(n)</td>
<td>Character whose ASCII value is n</td>
</tr>
<tr>
<td>A CLEAR</td>
<td>Clear all variables</td>
</tr>
<tr>
<td>P CLOSE f</td>
<td>Stop reading or writing a text file</td>
</tr>
<tr>
<td>A COLOR=n</td>
<td>Set lo-res color to n (0-15)</td>
</tr>
<tr>
<td>* COMMON f1,f2</td>
<td>Compile a share-variables file to disk</td>
</tr>
<tr>
<td>* COMPILER f1,f2</td>
<td>Load f1 (BAS), compile and save as f2 (COM)</td>
</tr>
<tr>
<td>A CONT</td>
<td>Continue a program</td>
</tr>
<tr>
<td>A COS(x)</td>
<td>Cosine of x in radians</td>
</tr>
<tr>
<td>P CREATE f</td>
<td>Create a subdirectory file</td>
</tr>
<tr>
<td>A DATA A$,x,y,z</td>
<td>Strings and values to be READ</td>
</tr>
<tr>
<td>A DEF FN A(x)=f(x)</td>
<td>Define a function</td>
</tr>
<tr>
<td>A DEL n,m</td>
<td>Delete program lines n through m</td>
</tr>
<tr>
<td>P DELETE f</td>
<td>Delete file f from disk</td>
</tr>
<tr>
<td>A DIM X(n)</td>
<td>Dimension a numerical array</td>
</tr>
<tr>
<td>A DIM A$(n)</td>
<td>Dimension a string array</td>
</tr>
<tr>
<td>A DRAW n AT l,j</td>
<td>Draw a hi-res shape at n at i,j</td>
</tr>
<tr>
<td>A END</td>
<td>Stop a program with no message</td>
</tr>
<tr>
<td>P EXEC f</td>
<td>Execute text file f from the keyboard</td>
</tr>
<tr>
<td>A EXP(x)</td>
<td>e (2.718289) to the xth power</td>
</tr>
</tbody>
</table>

### Additional Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A FLASH</td>
<td>Set flashing screen output (40-columns)</td>
</tr>
<tr>
<td>P FLUSH</td>
<td>Write buffer to disk without closing file</td>
</tr>
<tr>
<td>A FN</td>
<td>See DEF FN</td>
</tr>
<tr>
<td>A FOR X=n TO m</td>
<td>Let X=n, X=n+1... until X=m</td>
</tr>
<tr>
<td>P FRE</td>
<td>Free all available memory</td>
</tr>
<tr>
<td>A FRE(0)</td>
<td>Amount of free memory available</td>
</tr>
<tr>
<td>A GET A$</td>
<td>Wait for one-character user input</td>
</tr>
<tr>
<td>A GET X</td>
<td>Wait for one-number user input</td>
</tr>
<tr>
<td>A GOSUB n</td>
<td>Branch to subroutine at line n</td>
</tr>
<tr>
<td>A GOTO n</td>
<td>Branch to line n</td>
</tr>
<tr>
<td>A GR</td>
<td>View and clear lo-res page 1</td>
</tr>
<tr>
<td>A HCOLOR=n</td>
<td>Set hi-res color to n (0-7)</td>
</tr>
<tr>
<td>A HGR</td>
<td>View and clear upper hi-res page 1</td>
</tr>
<tr>
<td>A HGR2</td>
<td>View and clear full hi-res page 2</td>
</tr>
<tr>
<td>A HIMEM: n</td>
<td>Set highest memory address available</td>
</tr>
<tr>
<td>A HLIN n,m AT j</td>
<td>Draw a horizontal lo-res line</td>
</tr>
<tr>
<td>A HOME</td>
<td>Clear text screen to black</td>
</tr>
<tr>
<td>A HPLLOT i,j</td>
<td>Plot a hi-res point</td>
</tr>
<tr>
<td>A HPLLOT i,j TO n,m</td>
<td>Draw a hi-res line</td>
</tr>
<tr>
<td>A HTAB n</td>
<td>Position cursor at horizontal position n</td>
</tr>
<tr>
<td>A IF...THEN...</td>
<td>Logical &quot;if&quot; true, &quot;then&quot; execute</td>
</tr>
<tr>
<td>A IN#n</td>
<td>Take input from slot n</td>
</tr>
<tr>
<td>A INPUT X</td>
<td>(or A$) Wait for user input</td>
</tr>
<tr>
<td>A INPUT &quot;ABC&quot;;A$</td>
<td>(or X) Print &quot;ABC&quot; and wait for input</td>
</tr>
<tr>
<td>A INT(RND(1)+n)</td>
<td>Random integer 0 to n-1</td>
</tr>
<tr>
<td>A INVERSE</td>
<td>Set black-on-white text output</td>
</tr>
<tr>
<td>A LEFTS(A$,n)</td>
<td>First n characters of a string</td>
</tr>
<tr>
<td>A LEN(A$)</td>
<td>Number of characters in a string</td>
</tr>
<tr>
<td>A LET X=Y</td>
<td>Set X equal to Y (LET is optional)</td>
</tr>
<tr>
<td>A LIST</td>
<td>List a program from the beginning</td>
</tr>
<tr>
<td>A LIST n</td>
<td>List to line n</td>
</tr>
<tr>
<td>A LIST n-</td>
<td>List from line n</td>
</tr>
<tr>
<td>A LIST n,m</td>
<td>(or n,m) List lines n through m</td>
</tr>
<tr>
<td>A LOAD</td>
<td>Load a program from tape (obsolete)</td>
</tr>
<tr>
<td>P LOAD f</td>
<td>Load a file from disk</td>
</tr>
<tr>
<td>P LOCK f</td>
<td>Protect a disk file from alteration</td>
</tr>
<tr>
<td>A LOG(x)</td>
<td>Natural logarithm of x</td>
</tr>
<tr>
<td>A LOMEM: n</td>
<td>Set start-of-variables location</td>
</tr>
<tr>
<td>A MIDAS(A$,n,m)</td>
<td>m characters of AS, starting at n</td>
</tr>
</tbody>
</table>
COMMAND SUMMARY (continued)

A  NEW      Delete current program from memory
A  NEXT X   Define the end of a FOR-NEXT loop
A  NORMAL   Set normal white-on-black text output
A  NOT      Logical "not" in an IF statement
A  NOTRACE  Cancel TRACe
A  ON X GOSUB n,m,... GOSUB Xth line number
A  ON X GOTO n,m,... Branch to Xth line number
A  ONERRGOTO n Branch to line n if an error occurs
P  OPEN f   Begin READ or WRITE of a text file
A  OR       Logical "or" in an IF statement
A  PDL(n)   Value (0-255) of paddle n (0-3)
A  PEEK(n)  Memory value at location n
A  PLOT i,j  Plot a dot
A  POKE n,m Set location n to value m
A  POP      Cancel most recent GOSUB
A  POS(x)   Horizontal cursor position
P  POSITION f Locate READ or WRITE in a file
A  PRI n    Send output to slot n
P  PREFIX f Change default directory
P  PREFIX   Cancel current prefix
A  PRINT    Skip a text line
A  PRINT "ABC" Print characters within quotes
A  PRINT X  Print value of variable X
A  READ AS  (or X) Read a DATA string or value
P  READ f   Initiate reading a disk text file
A  RECALL X Retrieve array from tape (obsolete)
A  REM      Programmer's remark follows
P  RENAME f1,f2 Rename a file on disk
A  RESTORE  Reset pointer to first DATA statement
P  RESTORE f Retrieve strings and variables from file f
A  RESUME    Continue program where error occurred
A  RETURN   Branch back to statement after GOSUB
A  RIGHTS AS, n) Last n characters of a string
A  RND(0)   Repeat last random number
A  RND(n)   Random number (0 to 0.999999999)
A  ROT=n    Set rotation of a shape to n (0-64)
A  RUN      Execute a program from beginning
A  RUN n    Execute a program from line n
P  RUN f    Load and execute a program from disk
* RUN f    Load, compile & execute a program from disk
A  SAVE     Save a program to tape (obsolete)
A  SAVE f   Save a program f to disk
A  SCALE=n  Set scale for DRAW or XDRAW (0-255)
A  SCRN(i, j) Lo-res screen color at point i,j
A  SGN(x)   Sign of X (+1, -1 or 0)
A  SHLOAD   Load shape table from tape (obsolete)
A  SIN(x)   Sine of x in radians
A  SPCC(n)  n spaces in a PRINT statement
A  SPEED=n  Character output rate (0-255)
A  SQR(x)   Square root of x
A  STEP n   Increment-size in a FOR-NEXT loop
A  STOP     Stop program and print line number
P  STORE f  Store current variables as VAR file f
A  STORE X  Store an array on tape (obsolete)
A  STRS(x)  String equivalent of value x
A  TAB(n)   Position the cursor in a PRINT statement
A  TAN(x)   Tangent of x in radians
A  TEXT     Switch to text mode, cancel windows
A  THEN     Logical "then" in an IF statement
A  TO       See FOR and IPOINT
A  TRACE    Print line numbers as executed
A  UNLOCK f Cancel LOCK
A  USR(x)   Pass x to a machine subroutine
A  VAL($s)  Numeric value of a string
P  VERIFY f Verify that file f is on the disk
A  VLIN n,m AT i Draw a vertical lo-res line
A  VTAB n   Move the cursor to line n
A  WAIT i,j,k Insert a conditional pause
P  WRITE f  Initiate writing to a disk text file
A  XDRAW n AT i,j Draw in the opposite color
A  XPLOT    Execute file f
A  ?        Same as PRINT
MODIFYING APPLESOF T
VIA THE BEAGLE COMPILER
Actually, we're not going to tell you much on the next few pages, but we do want to give all the hackers, snoopers and other nice people out there a taste of how the Compiler works. Please consider this information as a freebie only—don't call Beagle Bros for help in analyzing the Compiler's functions.

HAVE FUN IMPROVING
(OR RUINING) APPLESOF T!
Programmers have always had a desire to modify the Applesoft interpreter to add more power and efficiency to a somewhat stale language. Since the interpreter is in ROM, it is a bit difficult to patch. The Beagle Compiler's interpreter, however, is in RAM, and you can change statements and functions at will.

The Jump Table starting on the next page goes from $9900 to $99FF and contains vectors (addresses) of each part of the Compiler that processes statements and functions. A few well-placed pokes from BASIC or machine language will "steer" commands to any area of memory you choose.

For example, the following program, when compiled, will make HOME act like HGR. From there, you're on your own.

10 POKE 39208, PEEO(39282): REM 39288, 89972
20 POKE 39209, PEEO(39283): REM 39289, 59973
21: REM 39288-29 IS THE ADDRESS FOR HOME
30 HOME: REM HOME NOW WORKS LIKE HGR
40 HCOLOR=1: HPLT(0,0) TO 279,191
Warning: Programs like the one above can quickly open up a whole B'n of up'ed!

G Plloribus
Spaggzettiecode

THE BEAGLE COMPILER
JUMP TABLE

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$9900</td>
<td>init</td>
<td>Initialize and run the program</td>
</tr>
<tr>
<td>$9902</td>
<td>run</td>
<td>RUN line number</td>
</tr>
<tr>
<td>$9904</td>
<td>clear</td>
<td>CLEAR</td>
</tr>
<tr>
<td>$9906</td>
<td>restore</td>
<td>RESTORE</td>
</tr>
<tr>
<td>$9908</td>
<td>on</td>
<td>ON COTO/GOSUB</td>
</tr>
<tr>
<td>$990A</td>
<td>goto</td>
<td>COTO</td>
</tr>
<tr>
<td>$990C</td>
<td>goub</td>
<td>GOSUB</td>
</tr>
<tr>
<td>$990E</td>
<td>return</td>
<td>RETURN</td>
</tr>
<tr>
<td>$9910</td>
<td>pop</td>
<td>POP</td>
</tr>
<tr>
<td>$9912</td>
<td>end</td>
<td>END (halts program)</td>
</tr>
<tr>
<td>$9914</td>
<td>let</td>
<td>assign value to simple numeric variable</td>
</tr>
<tr>
<td>$9916</td>
<td>for</td>
<td>FOR (set TO value and initialize loop)</td>
</tr>
<tr>
<td>$9918</td>
<td>step</td>
<td>STEP</td>
</tr>
<tr>
<td>$991A</td>
<td>rempt</td>
<td>evaluate and print a numeric value</td>
</tr>
<tr>
<td>$991C</td>
<td>strpr</td>
<td>evaluate and print a string value</td>
</tr>
<tr>
<td>$991E</td>
<td>spcc</td>
<td>SPC</td>
</tr>
<tr>
<td>$9920</td>
<td>tab</td>
<td>TAB</td>
</tr>
<tr>
<td>$9922</td>
<td>comma</td>
<td>comma function in PRINT statement</td>
</tr>
<tr>
<td>$9924</td>
<td>count</td>
<td>print a RETURN</td>
</tr>
<tr>
<td>$9926</td>
<td>text</td>
<td>TEXT</td>
</tr>
<tr>
<td>$9928</td>
<td>home</td>
<td>HOME</td>
</tr>
<tr>
<td>$992A</td>
<td>normal</td>
<td>NORMAL</td>
</tr>
<tr>
<td>$992C</td>
<td>inverse</td>
<td>INVERSE</td>
</tr>
<tr>
<td>$992E</td>
<td>flash</td>
<td>FLASH</td>
</tr>
<tr>
<td>$9930</td>
<td>next</td>
<td>NEXT</td>
</tr>
<tr>
<td>$9932</td>
<td>nextvar</td>
<td>NEXT statement with a variable</td>
</tr>
<tr>
<td>$9934</td>
<td>letstr</td>
<td>assign value to simple string variable</td>
</tr>
<tr>
<td>$9936</td>
<td>onerr</td>
<td>ONERR GOTO</td>
</tr>
<tr>
<td>$9938</td>
<td>prnum</td>
<td>PR#</td>
</tr>
<tr>
<td>$993A</td>
<td>isom</td>
<td>IN#</td>
</tr>
<tr>
<td>$993C</td>
<td>readn</td>
<td>READ a numeric value</td>
</tr>
<tr>
<td>$993E</td>
<td>readstr</td>
<td>READ a string value</td>
</tr>
<tr>
<td>$9940</td>
<td>getn</td>
<td>GET a numeric value (shouldn't be used)</td>
</tr>
<tr>
<td>$9942</td>
<td>gets</td>
<td>GET a string value</td>
</tr>
<tr>
<td>$9944</td>
<td>plot</td>
<td>PLOT</td>
</tr>
<tr>
<td>$9946</td>
<td>vlin</td>
<td>VLIN</td>
</tr>
<tr>
<td>$9948</td>
<td>hin</td>
<td>HIRON</td>
</tr>
<tr>
<td>$994A</td>
<td>if</td>
<td>IF</td>
</tr>
<tr>
<td>$994C</td>
<td>input</td>
<td>INPUT a numeric value</td>
</tr>
<tr>
<td>$994E</td>
<td>inputs</td>
<td>INPUT a string value</td>
</tr>
<tr>
<td>Address</td>
<td>Name</td>
<td>Function</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>----------</td>
</tr>
<tr>
<td>$9980</td>
<td>print a 'T'</td>
<td></td>
</tr>
<tr>
<td>$9982</td>
<td>CR</td>
<td></td>
</tr>
<tr>
<td>$9984</td>
<td>DRAW</td>
<td>DRAW</td>
</tr>
<tr>
<td>$9986</td>
<td>DRAW</td>
<td>DRAW with an AT</td>
</tr>
<tr>
<td>$9956</td>
<td>xdRAW</td>
<td>XDRAW</td>
</tr>
<tr>
<td>$9958</td>
<td>xdRAW</td>
<td>XDRAW with an AT</td>
</tr>
<tr>
<td>$995A</td>
<td>hplot</td>
<td>HIPLT</td>
</tr>
<tr>
<td>$995C</td>
<td>hplot</td>
<td>HIPLT</td>
</tr>
<tr>
<td>$9960</td>
<td>STOP</td>
<td>STOP</td>
</tr>
<tr>
<td>$9962</td>
<td>HCOLOR</td>
<td>HCOLOR</td>
</tr>
<tr>
<td>$9964</td>
<td>HTAB</td>
<td>HTAB</td>
</tr>
<tr>
<td>$9966</td>
<td>VTAB</td>
<td>VTAB</td>
</tr>
<tr>
<td>$9968</td>
<td>COLOR</td>
<td>COLOR</td>
</tr>
<tr>
<td>$996A</td>
<td>SPEED</td>
<td>SPEED =</td>
</tr>
<tr>
<td>$996C</td>
<td>POKE</td>
<td>POKE</td>
</tr>
<tr>
<td>$996E</td>
<td>CALL</td>
<td>CALL</td>
</tr>
<tr>
<td>$9970</td>
<td>HGR2</td>
<td>HGR2</td>
</tr>
<tr>
<td>$9972</td>
<td>HGR</td>
<td>HGR</td>
</tr>
<tr>
<td>$9974</td>
<td>SCALE</td>
<td>SCALE =</td>
</tr>
<tr>
<td>$9976</td>
<td>RCPT</td>
<td>RCPT =</td>
</tr>
<tr>
<td>$9978</td>
<td>USR</td>
<td>USR</td>
</tr>
<tr>
<td>$997A</td>
<td>PDL</td>
<td>PDL</td>
</tr>
<tr>
<td>$997C</td>
<td>PEER</td>
<td>PEER</td>
</tr>
<tr>
<td>$997E</td>
<td>LETINT</td>
<td>assign value to simple integer variable</td>
</tr>
<tr>
<td>$9980</td>
<td>LETARY</td>
<td>assign value to array</td>
</tr>
<tr>
<td>$9982</td>
<td>LEBRARY</td>
<td>assign value to string array</td>
</tr>
<tr>
<td>$9984</td>
<td>DIM</td>
<td>DIM</td>
</tr>
<tr>
<td>$9986</td>
<td>HIMEM</td>
<td>HIMEM</td>
</tr>
<tr>
<td>$9988</td>
<td>LOMEM</td>
<td>LOMEM</td>
</tr>
<tr>
<td>$998A</td>
<td>RESUME</td>
<td>RESUME</td>
</tr>
<tr>
<td>$998C</td>
<td>&amp; (use 2 of them)</td>
<td></td>
</tr>
<tr>
<td>$998E</td>
<td>WAIT</td>
<td>WAIT with 2 parameters</td>
</tr>
<tr>
<td>$9990</td>
<td>WAIT</td>
<td>WAIT with 3 parameters</td>
</tr>
<tr>
<td>$9992</td>
<td>DEF</td>
<td>DEF FN</td>
</tr>
<tr>
<td>$9994</td>
<td>BYTE</td>
<td>byte numeric constant</td>
</tr>
<tr>
<td>$9996</td>
<td>INTEGER</td>
<td>integer numeric constant</td>
</tr>
<tr>
<td>$9998</td>
<td>FR</td>
<td>floating point constant (packed)</td>
</tr>
<tr>
<td>$999A</td>
<td>LITERAL</td>
<td>string constant</td>
</tr>
<tr>
<td>$999C</td>
<td>GETWAR</td>
<td>get value of numeric variable</td>
</tr>
<tr>
<td>$999E</td>
<td>GETVAR</td>
<td>get value of string var</td>
</tr>
</tbody>
</table>

**Address** | **Name** | **Function** |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$99A0</td>
<td>OR</td>
<td>OR</td>
</tr>
<tr>
<td>$99A2</td>
<td>AND</td>
<td>AND</td>
</tr>
<tr>
<td>$99A4</td>
<td>ROPE</td>
<td>relational operator determined by the next byte; 0: &lt;, 3: =, 6: &lt;=, 9: &gt;, 12: &gt;=, 15: &gt;</td>
</tr>
<tr>
<td>$99A6</td>
<td>STRCOM</td>
<td>string relational operator (see above)</td>
</tr>
<tr>
<td>$99A8</td>
<td>ADD</td>
<td>+ (addition)</td>
</tr>
<tr>
<td>$99A9</td>
<td>SUB</td>
<td>- (subtraction)</td>
</tr>
<tr>
<td>$99AA</td>
<td>MUL</td>
<td>* (multiplication)</td>
</tr>
<tr>
<td>$99AB</td>
<td>DIV</td>
<td>/ (division)</td>
</tr>
<tr>
<td>$99B0</td>
<td>POWER</td>
<td>^ (exponentiation)</td>
</tr>
<tr>
<td>$99B2</td>
<td>NOT</td>
<td>NOT</td>
</tr>
<tr>
<td>$99B4</td>
<td>ASC</td>
<td>ASC</td>
</tr>
<tr>
<td>$99B6</td>
<td>CHR</td>
<td>CHR</td>
</tr>
<tr>
<td>$99B8</td>
<td>POS</td>
<td>POS</td>
</tr>
<tr>
<td>$99BA</td>
<td>LEN</td>
<td>LEN</td>
</tr>
<tr>
<td>$99BC</td>
<td>LEFTS</td>
<td>LEFTS</td>
</tr>
<tr>
<td>$99BE</td>
<td>RIGHTS</td>
<td>RIGHTS</td>
</tr>
<tr>
<td>$99C0</td>
<td>MID2</td>
<td>MID3 with 2 parameters</td>
</tr>
<tr>
<td>$99C2</td>
<td>MID3</td>
<td>MID3 with 3 parameters</td>
</tr>
<tr>
<td>$99C4</td>
<td>STR</td>
<td>STRS</td>
</tr>
<tr>
<td>$99C6</td>
<td>AHB</td>
<td>AHB</td>
</tr>
<tr>
<td>$99C8</td>
<td>MID</td>
<td>MID</td>
</tr>
<tr>
<td>$99CA</td>
<td>SGN</td>
<td>SGN</td>
</tr>
<tr>
<td>$99CC</td>
<td>INT</td>
<td>INT</td>
</tr>
<tr>
<td>$99CE</td>
<td>VAL</td>
<td>VAL</td>
</tr>
<tr>
<td>$99D0</td>
<td>NEG</td>
<td>negate</td>
</tr>
<tr>
<td>$99D2</td>
<td>CONCAT</td>
<td>+ (string concatenation)</td>
</tr>
<tr>
<td>$99D4</td>
<td>SCR</td>
<td>SCRN</td>
</tr>
<tr>
<td>$99D6</td>
<td>FRE</td>
<td>FRE</td>
</tr>
<tr>
<td>$99D8</td>
<td>SQR</td>
<td>SQR</td>
</tr>
<tr>
<td>$99DA</td>
<td>LOG</td>
<td>LOG</td>
</tr>
<tr>
<td>$99DC</td>
<td>EXP</td>
<td>EXP</td>
</tr>
<tr>
<td>$99DE</td>
<td>COS</td>
<td>COS</td>
</tr>
<tr>
<td>$99E0</td>
<td>SIN</td>
<td>SIN</td>
</tr>
<tr>
<td>$99E2</td>
<td>TAN</td>
<td>TAN</td>
</tr>
<tr>
<td>$99E4</td>
<td>ATN</td>
<td>ATN</td>
</tr>
<tr>
<td>$99E6</td>
<td>GETVAR</td>
<td>get value of numeric array</td>
</tr>
<tr>
<td>$99E8</td>
<td>GETSARY</td>
<td>get value of string array</td>
</tr>
<tr>
<td>$99EA</td>
<td>FN</td>
<td>FN (user-defined function)</td>
</tr>
<tr>
<td>$99EC</td>
<td>SAVES</td>
<td>save code pointer if RESUME exists</td>
</tr>
<tr>
<td>$99E0</td>
<td>ERROR</td>
<td>process an error</td>
</tr>
<tr>
<td>$99F0</td>
<td>INPUT</td>
<td>prepare for a numeric or string input</td>
</tr>
<tr>
<td>$99F3-$99FF</td>
<td></td>
<td>reserved—you touch and we call the cops!</td>
</tr>
</tbody>
</table>

END OF TABLE
USER-AVAILABLE ROUTINES
The following routines are available to help you in writing assembly language routines that will interface with Applesoft:

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$98E8</td>
<td>MOVSTR</td>
<td>Moves a string (no length byte, must end with 0 byte) to string space which has already been allocated with &quot;GETSPA&quot;. On entry, A = LSB of string address and X = MSB of string address.</td>
</tr>
<tr>
<td>$98EB</td>
<td>BYTE</td>
<td>Evaluates a numeric parameter and verifies that it is a byte value (0-255). Anything else gives ILLEGAL QUANTITY. The value is returned in both A and X.</td>
</tr>
<tr>
<td>$98EE</td>
<td>PRNUM</td>
<td>Prints a numeric value. Set the carry flag if the value is floating point (in the FAC). If the value is integer, clear the carry flag and the value should be in X (LSB) and A (MSB).</td>
</tr>
<tr>
<td>$98F1</td>
<td>ERROR</td>
<td>Calls ERROR to report an error. X should contain the number of the error message (see Apple BASIC Reference Manual). If ONERR is not enabled, the program will stop and print the error message.</td>
</tr>
<tr>
<td>$98F4</td>
<td>GETSPA</td>
<td>Allocates space in the string area. A = length of the string. One additional byte is allocated because the first byte contains the length of the string. Put the string where $F6,5F7 points.</td>
</tr>
<tr>
<td>$98F7</td>
<td>PRSTR</td>
<td>Prints the string pointed to by $76,5F7.</td>
</tr>
<tr>
<td>$98FA</td>
<td>PROC</td>
<td>Processes the next statement. A jump to this location is done at the end of every BASIC statement. If you are replacing a statement, your code should end with this.</td>
</tr>
<tr>
<td>$98FD</td>
<td>EVAL</td>
<td>Evaluates a numeric or string parameter.</td>
</tr>
</tbody>
</table>

VARIABLES
Variables are indicated by a byte value ($00-FF). The values for the variables are accessed by using the byte value as an index into the tables at the addresses indicated by the following pointers:

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| vtype ($7B)   | Variable type:
|               | bit 7 = 1 if array, 0 if not
|               | bit 6 = 1 if string, 0 if numeric
|               | bit 5 = 1 if FN (user-defined function)
|               | bit 4 = 1 if integer
|               | bits 0-3 = number of dimensions if array |

The following information depends upon the type of variable involved:

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP</td>
<td>Floating point, LSB=least significant byte, MSB=most significant byte.</td>
</tr>
</tbody>
</table>
| vval1 (7A)    | Array: LSB of address of array header
|               | String: LSB of address of string
|               | FP: Non-zero value indicates this is floating point type, 1st byte of packed FP value
|               | Integer: This value is zero if variable has an integer value |
| vval2 (7C)    | Array: MSB of address of array header
|               | String: MSB of address of string
|               | FP: 2nd byte of FP value
|               | Integer: LSB of integer value |
| vval3 (7E)    | Array: LSB of address of array
|               | FP: 3rd byte of FP value
|               | Integer: MSB of integer value |
| vval4 (80)    | Array: MSB of address of array
|               | FP: 4th byte of FP value |
| vval5 (82)    | Array: Number of dimensions, 0 if not dimensioned yet
|               | FP: 5th byte of FP value |

If variable A$ has a variable index of 2, then the following code would assign A$ the string pointed to by PTR:

```assembly
LDY #2   ;variable A$
LDA PTR
STA ($7A),Y ;store LSB of address
LDA PTR+1
STA ($7C),Y ;store MSB of address
```
INDEX

Address of program ............ 16
Ampersand (&) .............. 20
Appendix machine code ....... 24
Apple software .............. 7,8,30
Apple software, changing .... 13,34
Backups ...................... 3
Break at SXXX message ....... 14,23
Chain .......................... 18
Changing programs .......... 13
Commands ..................... 30
COM files ...................... 8,11
COMMON command ........... 18,19
COMPILE command ........... 11,18,19
COMPILE file ................. 6,26,27
COMPILE SYSTEM file ....... 6,26,27
Control-C ..................... 14
CVR files ...................... 19
DOS 3.3 ......................... 4
Errors ......................... 23-25
<>, <#>, <A> ................. 24
Error at SXXX message ....... 23
File Type Mismatch ......... 25
Line nos., converting ....... 23
No Buffers Available ....... 25
Not a Compiled Program .... 25
Program Too Large .......... 25
Syntax Error .................. 25
FAST.HPLOT program ....... 29
Giving programs away ...... 6
G.P.L.E ......................... 5
HPLOT speed .................. 29
INT files ....................... 8,11
INPUT.ANYTHING program .. 29
Installing the Compiler ....... 8,27
Joystick problems ......... 29
Licensed Compiler .......... 6
MENU program ............... 28
NOTES program .............. 7
Paddle problems .......... 29
PRINT.LINES program ....... 23
PROGRAM WRITER .......... 5
Relocating programs ....... 16
RESTORE command ......... 19
Running programs .......... 10,12,15
Saving programs .......... 11
Saving space ............... 27
Selling programs .......... 6
SLOW.PEL program ......... 23
Speed, maximum .......... 17
STARTUP program .......... 26,27
STORE command .......... 19
TESTPROGRAM ............... 9
Uncompilable commands .... 9
Worms, can of ............ 34

Disclaimer of All Warranties and Liabilities

Even though the software described in this manual has been tested and released, neither Beagle Bros nor its software suppliers make any warranty or representation, either express or implied, with respect to this manual, the software and/or the diskette: their quality and performance, merchantability, or fitness for any particular purpose. As a result, the diskette, software and manual are sold "as is" and you, the purchaser, are assuming the entire risk as to their quality and performance. In no event will Beagle Bros or its software suppliers be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the diskette, software, or manual, even if they have been advised of the possibility of such damages. In particular, they shall have no liability for any programs or data stored in or used with Beagle Bros products, including the costs of recovering or reproducing those programs or data. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you.

About ProDOS: This product includes software, ProDOS, licensed from Apple Computer, Inc. Apple makes no warranties, either express or implied, regarding the enclosed software package, its merchantability or fitness for any purpose. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you.
Using The Beagle Compiler With Extended Memory

MEMORY TO BURN!
The latest version of the Beagle Compiler takes advantage of two types of extended memory, letting you automatically store strings and arrays outside of main (48K) memory. This gives you room for larger compiled programs.

Use **AUX.SLOT.SYSTEM** instead of **COMPILER.SYSTEM** with:
- Apple IIc or 128K IIe
- Apple IIgs (only 128K will be accessed)
- Applied Engineering RamWorks™ (all models)
- Checkmate MultiRam™ (all models)

Use **APPLEMEM.SYSTEM** instead of **COMPILER.SYSTEM** with:
- Apple Memory Expansion Card
- Applied Engineering RamFactor
- Flipster

To use one of these **SYSTEM** files, transfer it to another disk along with the file **FRODO.S**. Then boot that disk. The file you want installed during bootup (**COMPILER.SYSTEM**, **AUX.SLOT.SYSTEM** or **APPLEMEM.SYSTEM**) must be the first **SYS** file whose name ends in "SYSTEM" on the disk.

To prevent the need to transfer files from the Beagle Compiler disk, you could simply rename the two **SYSTEM** files that you don't want to use. Use names that don't end in "SYSTEM".

Note: The Beagle Compiler's **STARTUP** program, if used, will report "**COMPILER.SYSTEM INSTALLED**" regardless of which of the three **SYSTEM** files has been installed.

Another Note: **APPLEMEM.SYSTEM** uses memory more efficiently than **AUX.SLOT.SYSTEM**, which does not use about 28% of the memory available. (Not that you have a choice—the type of hardware you have determines which **SYSTEM** file you must use.)

NO STORE OR RESTORE
Sorry, ProDOS's **STORE** and **RESTORE** commands will not work with **APPLEMEM.SYSTEM** or **AUX.SLOT.SYSTEM**.

IMPORTANT RAM DISK NOTES
When you install **APPLEMEM.SYSTEM** or **AUX.SLOT.SYSTEM**, anything stored in a RAM disk will be WIPED OUT. Be sure all of your files are copied to a real disk first!

**APPLEMEM.SYSTEM** cannot be used with a RAM disk.
**AUX.SLOT.SYSTEM** can be configured to work with a RamWorks or MultiRam RAM disk. Here's what you do:
1. With **BASIC.SYSTEM** installed (not one of the Compiler's **SYSTEM** programs), run the **CONFIGURE.BANKS** program and specify the minimum and maximum memory banks that will be used by the Compiler. Press **RETURN** to save a reconfigured version of **AUX.SLOT.SYSTEM**.
   
   Be sure the range you have selected does not overlap the memory that will be used by your RAM disk.*

2. Use the partition program that came with your RAM disk to limit the memory your RAM disk uses. Be sure this memory does not overlap the memory used by the Compiler.*

   *Most RAM disks don't use Bank 0. You can probably skip step 2 by configuring the Compiler to use only Bank 0 (in step 1, set both minimum and maximum to 0).

MAXIMUM ARRAY SIZES
With **APPLEMEM.SYSTEM** or **AUX.SLOT.SYSTEM** installed, arrays may be dimensioned to the following maximum sizes:

<table>
<thead>
<tr>
<th>Array type</th>
<th><strong>AUX.SLOT.SYSTEM</strong></th>
<th><strong>APPLEMEM.SYSTEM</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>9419</td>
<td>13106</td>
</tr>
<tr>
<td>Integer</td>
<td>23550</td>
<td>32766</td>
</tr>
<tr>
<td>String</td>
<td>15699</td>
<td>21844</td>
</tr>
</tbody>
</table>

PROGRAM SPEED
Your compiled programs may run slightly slower under **AUX.SLOT.SYSTEM** and **APPLEMEM.SYSTEM**. The difference in speed will depend mainly upon how much string manipulation your Applesoft program does.

HOW MUCH FREE MEMORY?
After installing **AUX.SLOT.SYSTEM** or **APPLEMEM.SYSTEM**, run the Applesoft program **HOW.MUCH** to see how much free memory you have for variables. Get ready to see some big numbers!