PROGRAMS IN PIECES

ProDOS makes it easy to create long programs that run in segments. This excerpt from *ProDOS Inside and Out* shows you how it's done.

Normally, BASIC programs are stored on disk using the SAVE command and brought back into memory using LOAD or RUN (or "—"). The program reserves a certain portion of the available RAM memory. Most of the remaining RAM memory is still available for other uses by Applesoft and BASIC.SYSTEM.

There is a continual battle fought between the programmer and his computer: the battle for space. The programmer always wants more space to expand the performance of his creation; the computer may or may not have room to grant his wish. Eventually, if enough bells and whistles are added to a program, size constraints start to become noticeable. When the PROGRAM TOO LARGE error occurs, the battle enters a new level.

BASIC.SYSTEM has three commands to facilitate managing programs in less memory than they would require if handled as a single large entity. These commands are CHAIN, STORE, and RESTORE.

DIVIDE AND CONQUER

Sometimes you get really involved in adding improvements to a program, to the point where the program starts to outgrow the computer. With Applesoft BASIC under ProDOS and BASIC.SYSTEM, you have a maximum of about 34K bytes of space to program; this can be much less due to space you need to store your variables, any machine-language support routines, etc. At the point where you run out of room, you have to make a decision: Reduce the memory requirements of your single program by improving its size efficiency or removing features, or split the program into smaller subsections. Examples of the first approach are removing unneeded REM statements, combining program lines, defining variables to use a minimum of space (including use of arrays), and so forth. This will provide some help in many cases.

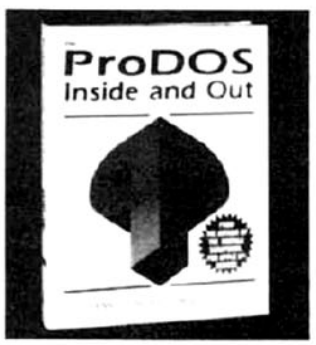
But a truly big program, such as a database, will require that you split up the program's functions. A database program will have routines for setting up the database format, entering data, storing the data on disk, retrieving the data, as well as an extensive set of routines to print reports, which opens a whole extra set of data to store for report and printer formats. Such a program can easily fill the Apple memory by itself, without leaving any room for the data. And without room for the data, why bother with the program?

If you think about it, though, most of the sections of the database program don't have overlapping responsibilities. You can only work with one section of the program at a time. For example, if you are entering names and addresses, you aren't going to be setting up a printer format at the same time, since that is a separate operation. Why not, then, put the sections of the program that you are not using on disk out of the way, and call them in only when needed?

Thinking About Trade-Offs

The disadvantage of having sections of the program on disk is the delay encountered in reading a new section you need into memory. There's also the irritation of having to keep the disk with all the program sections handy. The delay factor is not a big concern if the program is broken up into sections intelligently, especially since ProDOS only takes a few seconds to load a file at BASIC.SYSTEM's request. And, using ProDOS' ability to identify disks and files by name, the program can always check to see if the right disk and program are present. What you get for all this irritation is the ability to easily run a bigger program in sections than you could possibly fit into memory at one time. Also, you can keep more of the memory free as space for your variables and support routines.

One other problem: the Applesoft RUN command clears all variables from memory when starting execution of the program. BASIC.SYSTEM's RUN command does the same thing. It's really going



to slow things down if we have to save all of our variables before we can RUN the next section, which will then have to reread all of the data. It would be better if we could keep our diligently entered variable data in memory and just swap BASIC program sections.

Linking The Segments

There is, happily, a BASIC.SYSTEM command called CHAIN that lets you do this. It has the syntax:

```
CHAIN pathname,@ln,Sn,Dn
```

This command is similar to the BASIC.SYSTEM RUN command, except that it does not clear the current variables. A variable defined in a program that executes a second program via CHAIN can be referenced by the second program using the same name and will have the same value.

The programmer always wants more space to expand the performance of his creation.

As an example, let's think of a "first" program (shown in **Listing 1**) saved to disk as CHAIN.DEMO.1, and a second program (shown in **Listing 2**) saved to the same disk as CHAIN.DEMO.2. If we RUN CHAIN.DEMO.1, it will set the value of the variable NAME$ to JOHN DOE, and then execute CHAIN.DEMO.2 via the CHAIN command. CHAIN.DEMO.2 enters memory with the value of variable NAME$ held in memory intact, and it uses that value when printing the statement in **line 30**. If you substitute RUN for CHAIN in **line 60** of CHAIN.PART.1 to use the normal BASIC.SYSTEM RUN command, you'll see that NAME$ is cleared to an empty string before being printed in **line 30** of CHAIN.DEMO.2.

This same approach can be used for a much larger program to break it into manageable pieces that will fit in memory. All of the segments may use common names for specific variables and CHAIN to the required segment at will.

Note: Although this demo program works, Peter Meyer reported a bug in CHAIN in the June 1985 issue of *Call-A.P.P.L.E.*, in "A Bug in the ProDOS CHAIN Command" on page 30. Briefly, the first variable defined in a program may be lost during a CHAIN operation. He recommends defining an unused "dummy" variable at the beginning of any program that issues the CHAIN command.

A further cure for the problem was published in the July 1987 issue of *Byte* magazine on pages 305-310. The bug in CHAIN in BASIC.SYSTEM version 1.1 can be corrected by changing a byte within BASIC.SYSTEM before using CHAIN and restoring it immediately after the CHAIN operation is complete (failure to restore the byte was reported to cause problems with RESTORE). The information in *Byte* includes a permanent modification to BASIC.SYSTEM, but the BASIC solution is to use:

```
IF PEEK(49149) = 1 THEN POKE 41859,3: REM fix CHAIN
```

immediately before the program line containing the CHAIN command. PEEK(49149) verifies that we are dealing with BASIC.SYSTEM before we start changing things. Next, CHAIN is issued, and one of the first actions of the program CHAINed to should be to use:

```
IF PEEK(49149) = 1 THEN POKE 41859,7: REM restore
 byte
```

to set BASIC.SYSTEM back to normal.

Variables Using Memory

One major use of the remaining memory by Applesoft during the execution of most BASIC programs is for storage of variables. If you are familiar with working with BASIC programs, you know that many programming statements can be dedicated to defining initial values for variables, and that all variables defined in the program are stored as a part of the program itself. When the BASIC program is executed, these values are moved into the variable space area of RAM, and the value of each variable is associated with an area of RAM memory used to contain its representation. In addition, a certain amount of bookkeeping information is used to keep track of the location and length of string variables.

Single-Entry Bookkeeping

One way to decrease the size of a program, although impossible, would be to eliminate the double-bookkeeping of having variables defined in the program and also in the RAM area assigned for the variables. You could create most (or all) of the variables to be used, save those variables out to disk in a compact form, and then erase the definitions from the BASIC program. Then, in our ideal world, the program could later reload the set of initialized variables when needed. BASIC.SYSTEM provides for such a system using two commands, STORE and RESTORE.

The STORE command compresses ("tokenizes") the current defined set of BASIC variables within the variable space area of RAM (not within the program itself) into a block and saves that block to disk under a specified filename. The command syntax is:

```
STORE pathname,Sn,Dn
```

The command can be used in the immediate mode, but would normally be used in conjunction with a BASIC program to place the current set of defined values within the variable space into a disk file of type VAR (for variable). This is one way to store data from a BASIC program. Another, more conventional, manner is with text files. Binary files are also used, usually to store data that exists as a raw image of a section of memory.

Abusing STORE and RESTORE makes a program very hard to read and modify.

A variable file created with the STORE command can be recalled into memory using the RESTORE command, so the data can be reused:

```
RESTORE pathname,Sn,Dn
```

Variables stored with the original command can now be accessed with the same variable names; that is, the value of an arbitrary variable named MI$(11) saved with the STORE command can again be referenced as MI$(11) after RESTORE is used to reload the same variable file. In fact, you *must* use the same variable name for a defined value that was in effect when STORE was used to create a file.

In addition, the VAR file loaded by RESTORE will replace all currently defined variables in memory, including D$ = CHR$ (4). If you want D$ to have the same value in your program after RESTORE, you must save it as part of the file when using STORE, or define it again after the RESTORE command has been issued.

One valid application of STORE and RESTORE is to save space, as previously discussed. The variable file uses disk space efficiently (*very* efficiently versus text files), since the data is compressed before it's saved on disk. One disadvantage to the compression is that there may be a brief pause between the time the STORE command is issued and when the variable file can be created because of the time spent preparing the variable data for storage. All the Applesoft statements defining variables in the program can then be removed

(saving space within the program) and replaced with an initial BASIC.SYSTEM RESTORE command to load in the predetermined values.

A second application is the ability to define two or more different sets of variables for a single program, save each under a different filename, and load the desired set depending on conditions tested by the program.

STORE And Debugging

A third and less obvious use is in debugging a BASIC program. When a program is stopped with an END statement (or by encountering the last statement in the program's flow), with a Control-C (Break) typed from the keyboard, or by an error, the current values of the variables remain in RAM in the variable space. Any changes to the program to change an errant line will cause Applesoft to clear the current values of the variables. STOREing the variables will allow you to RESTORE them for later examination.

Since a change in a program line forces Applesoft to update its accounting for the location of the start and end of the program, Applesoft clears the current variable space and starts anew when a program line is changed or reentered.

The variable file uses space efficiently since the data is compressed before it's saved on disk.

Resetting the Apple also causes the variable space to be cleared for BASIC.SYSTEM's initialization routines. To allow yourself to edit a line and still retain the variable definitions, you can issue a STORE command, such as STORE DEBUG.1, to save the current variable values, edit the line, and then use a RESTORE DEBUG.1 to read in the variable set. Actually, you could save the state of the variables at several different stages of the program by using different names, and RESTORE each set by name if you like.

Eschew Obfuscation

Abusing STORE and RESTORE makes a program very hard to read and modify. Since the variable definitions can be removed completely from a program by using variable files, the (usually nontrivial) documentation value of the definitions is no longer present in such a program. The writing of undocumented programs is a bad practice — we will say no more.

A STORE Demo

For an example of the STORE command, try the program in **Listing 3**. RUN it, and the program will create a file named STRING.DATA of type VAR (verify this by using the CAT or CATALOG command, if you like).

The file can be read in by the program in **Listing 4**. When this program is run, note that the values printed for the strings before the RESTORE command is issued are all the null string. After RESTORE, the values are the same as those values STOREd in the file STRING.DATA by the previous program.

Note also the difference between this command and the CHAIN method of holding variables for use by another program. CHAIN keeps the values of the variables in RAM memory only; if power is lost to the computer or memory is otherwise cleared, the value of these variables is lost, just as a BASIC program is lost from memory under similar circumstances. STORE and RESTORE use the disk to store their variable tables and values, so that the record of the variables' values is as permanent as the disk file.

LISTING 1: CHAIN.DEMO.1

```
10  REM *** CHAIN DEMO (PART 1 OF TWO) ***
20 D$ =  CHR$ (4): REM CTRL-D
30  TEXT : HOME
50 NAME$ = "JOHN DOE"
60  PRINT D$;"CHAIN CHAIN.DEMO.2"
```

END OF LISTING 1

LISTING 2: CHAIN.DEMO.2

```
10  REM *** CHAIN DEMO (PART 2 OF TWO) ***
20  TEXT : HOME : PRINT "THE VALUE OF 'NAME$' IS:"
30  PRINT : PRINT  SPC( 10); CHR$ (34);NAME$; CHR$ (34)
40  VTAB 22: END
```

END OF LISTING 2

LISTING 3: STORE.DEMO

```
10  REM *** VARIABLE FILE DEMO PART 1 ***
20  TEXT : HOME :D$ =  CHR$ (4)
30  FOR I = 1 TO 5
40 A$(I) = "THIS IS STRING #" +  STR$ (I) + "."
50  NEXT I
60  PRINT D$;"STORE STRING.DATA"
99  END
```

END OF LISTING 3

LISTING 4: RESTORE.DEMO

```
10  REM *** VARIABLE FILE DEMO PART 2 ***
20  TEXT : HOME :D$ =  CHR$ (4)
30  PRINT "BEFORE 'RESTORE':"
40  PRINT : GOSUB 1000
50  VTAB 22: PRINT "PRESS A KEY: ";
60  GET A$: PRINT
100  HOME
110  PRINT "AFTER 'RESTORE':"
120  PRINT D$;"RESTORE STRING.DATA"
130  PRINT : GOSUB 1000
140  VTAB 22: END
199  END
1000  REM PRINT STRING VALUES.
1010  FOR I = 1 TO 5
1020  PRINT  SPC( 5);"A$(";I;") = "; CHR$ (34);A$(I);
      CHR$ (34)
1030  NEXT I
1040  RETURN
```

END OF LISTING 4