Addendum to the

Apple Pascal

Language Reference Manual

TABLE OF CONTENTS

APPLE Part #Ø31-Ø1Ø1-ØØ

INTRODUCTION

This document is an addendum to the Apple Pascal Language Reference Manual. Most of the items described are features that have been added to the system since the printing of the manual. Corrections to the manual are also included.

ONE-DRIVE STARTUP

The one-drive startup described on pages 148-15Ø of the Apple Pascal Language Reference Manual is not correct. Instead a one-drive startup works as follows:

Insert the diskette marked APPLE3: in the disk drive. Close the door to the disk drive and turn on the computer. The message

```
APPLE II
```

will appear on the screen and the disk drive's IN USE light will come on. The disk drive emits a whirring sound, lights up for a second or so with a screenful of black at-signs (@) on a white background, and then goes black again. Next the following message is displayed:

```
INSERT BOOT DISK WITH SYSTEM.PASCAL
ON IT. THEN PRESS RESET
```

To complete the booting process, insert APPLEØ: and then press RESET. After about 1Ø seconds, this message appears in the center of the screen:

```
WELCOME APPLEØ, TO APPLE II PASCAL 1.1
BASED ON USCD PASCAL II.1
CURRENT DATE IS 14-AUG-8Ø

(C)  APPLE COMPUTER INC. 1979, 198Ø
(C)  U.C. REGENTS 1979
```

The date may be different. The top of the screen will contain the Command level prompt line.

CHAINING PROGRAMS

Version 1.1 provides a new UNIT called CHAINSTUFF in the SYSTEM.LIBRARY file. This unit allows one program to "chain to" another program. This means that the first program specifies the second one by giving its filename; the system then executes the second program as soon as the first one terminates normally.

The CHAINSTUFF unit also allows the first program to pass a STRING value to the second program; note that this allows almost any information to be passed, since the string can be a filename and can thus specify a communications file containing almost anything. CHAINSTUFF also allows a program to turn the system swapping feature on and off. (System swapping is a new feature described in the Addendum to the Apple Pascal Operating System Reference Manual.)

CHAINSTUFF provides these capabilities in the form of five procedures named SETCHAIN, SETCVAL, GETCVAL, SWAPON, and SWAPOFF. To use these procedures, the program must have a USES declaration immediately after the program heading:

```
PROGRAM STARTER;
USES CHAINSTUFF;
...
```

The SYSTEM.LIBRARY file must be on line when the program is compiled and executed.

THE SETCHAIN PROCEDURE

The SETCHAIN procedure call has the form

```
SETCHAIN ( NEXTFILE )
```

where NEXTFILE is a STRING value (up to 23 characters). It should be either the name of a code file, or the name of an exec file with the prefix EXEC/. As soon as the program terminates normally, the system will proceed to execute the file whose name is the value of NEXTFILE.

The file is executed exactly as if the X(ecute command had been used; thus it is not necessary to supply the suffix .CODE for a code file or .TEXT for an exec file.

If the program is halted because of any run-time error, the chaining does not occur. Note that this includes a halt caused by the HALT procedure. However a termination caused by the EXIT procedure is considered a normal termination and the chaining will work.

THE SETCVAL PROCEDURE

The SETCVAL procedure call has the form

```
SETCVAL ( MESSAGE )
```

where MESSAGE is a STRING value (up to 80 characters). SETCVAL stores the MESSAGE in a system location called CVAL, where it can be picked up by another program.

THE GETCVAL PROCEDURE

The GETCVAL procedure call has the form

```
GETCVAL ( MESSAGE )
```

where MESSAGE is a STRING variable whose value is altered by GETCVAL. GETCVAL picks up the current value of CVAL from the system and stores it in the MESSAGE variable. Note that if CVAL has not been set by another program (using SETCVAL), then the value of CVAL is a zero-length string. Once CVAL has been set, it remains set to the same STRING value until it is changed or the system is reinitialized or rebooted.

SWAPON AND SWAPOFF

These procedures have no parameters. They allow a program to turn the system swapping feature on or off upon termination of the program (before chaining to another program).

AN EXAMPLE OF CHAINING

Suppose that a diskette named GAMES: contains a collection of game programs whose code files have the following names:

```
CHESS.CODE
CHECKERS.CODE
BLASTOFF.CODE
GOMOKU.CODE
BKGAMMON.CODE
BLACKJCK.CODE
HEARTS.CODE
SPROUTS.CODE
```

The user could use the Filer to display a list of filenames on the GAMES: diskette, then return to the Command level and use X(ecute to execute a selected program. Instead, however, you can write a "front-end" program to display a menu of all the available games; the user chooses one by typing a number, and the front-end program chains to the selected game program:

```
PROGRAM FRONT;
USES CHAINSTUFF;

VAR GAMENUM: INTEGER;

BEGIN
(*Display a greeting*)
  WRITELN('WELCOME TO GAMES!');
  WRITELN;
(*Display the menu*)
  WRITELN('SELECT A GAME FROM THE LIST BY TYPING ITS NUMBER:');
  WRITELN;
  WRITELN('1 -- CHESS');
  WRITELN('2 -- CHECKERS');
  WRITELN('3 -- BLASTOFF');
  WRITELN('4 -- GOMOKU');
  WRITELN('5 -- BACKGAMMON');
  WRITELN('6 -- BLACKJACK');
  WRITELN('7 -- HEARTS');
  WRITELN('8 -- SPROUTS');
  WRITELN;
(*Get a number from the user*)
  WRITE('TYPE A NUMBER FROM 1 THROUGH 8, THEN PRESS RETURN: ');
  READLN(GAMENUM);
(*Make sure the number is valid*)
  WHILE NOT (GAMENUM IN [1..8]) DO BEGIN
    WRITE('NUMBER MUST BE FROM 1 THROUGH 8 -- TRY AGAIN: ');
    READLN(GAMENUM)
  END;
```



```
(*Set chaining to filename of selected game*)
  CASE GAMENUM OF
    1:  SETCHAIN('GAMES:CHESS');
    2:  SETCHAIN('GAMES:CHECKERS');
    3:  SETCHAIN('GAMES:BLASTOFF');
    4:  SETCHAIN('GAMES:GOMOKU');
    5:  SETCHAIN('GAMES:BKGAMMON');
    6:  SETCHAIN('GAMES:BLACKJCK');
    7:  SETCHAIN('GAMES:HEARTS');
    8:  SETCHAIN('GAMES:SPROUTS')
  END
END.
```

There are several advantages to this. For one thing, the GAMES: diskette may have many other files besides the actual game programs, and this could be confusing to the user. For another, the FRONT program menu gives full and correct names for the games, since it is not limited to 8-character names; thus it lists BACKGAMMON instead of BKGAMMON.

Many game programs ask the user to type in her name, so it can be used in messages and prompts from the program. You could also have the FRONT program get the user's name and pass it to the selected game program. To do this, the FRONT program can declare a STRING variable, NAME, and then include the following lines either just before or just after the CASE statement:

```
(*Get user's name and store it in CVAL*)
WRITE('TYPE YOUR NAME, PLEASE:  ');
READLN(NAME);
SETCVAL(NAME)
```

Now a game program that uses the user's name can obtain it by having its own STRING variable named (for example) UNAME, and then calling GETCVAL:

```
GETCVAL(UNAME)
```

Incidentally, if the FRONT program's codefile is placed on the boot diskette and given the name SYSTEM.STARTUP, the FRONT program will be run automatically as soon as the system is started.

THE "V" OPTION

When a procedure or function has a VAR parameter of type STRING, the actual parameter in each call to the procedure or function is checked by the Compiler to make sure that its maximum length is not less than the maximum length of the formal parameter. In the previous version of the Compiler, there was no such checking.

This checking is controlled by the V option:

Default value: V+

(*$V+*) Turns checking on.

(*$V-*) Turns checking off.

The U- option also turns this checking feature off. Note that if checking is off and the maximum length of the actual parameter is less than the maximum length of the formal parameter, it is possible for the procedure or function to alter bytes of data that are beyond the end of the actual parameter variable. This does not cause a run-time error, but does cause unpredictable results.

THE "SWAPPING" OPTION

With the S+ option of the Compiler, the extra space available for symbol-table storage is about 53ØØ words (compared to about 39ØØ in the previous version). Going from S+ to S++ gives about 15ØØ words more.

STRINGS

The manual for Version 1.1 states (on page 9) that one string is "greater than" another if it would come first in an alphabetic list of strings. This is backwards: it should state that one string is "less than" another if it would come first in an alphabetic list of strings.

TURTLEGRAPHICS

The system will automatically return to text mode if a program terminates while in graphics mode. This applies to both normal termination and to a halt caused by a run-time error.

Also, the manual erroneously states that the TURTLEGRAPHICS procedures will accept REAL parameters for X,Y coordinates, and convert them to INTEGER values. Actually, X,Y coordinates are parameters of type INTEGER and if a REAL value is supplied an error results.

ERROR MESSAGES

The following new compiler error messages have been added:

175: Actual parameter max string length < var formal max length

4Ø8: (*$S+*) Needed to compile units

MEMORY SPACE FOR COMPILER

When compiling a very large program, it is possible for the compiler to run out of memory space. There are several remedies to try when this happens:

Use the command-level swapping option to get 11ØØ words of additional memory space.

Use the compiler swapping option (*$S+*) or if necessary, (*$S++*).

If the program uses "include" files, use the Filer's M(ake command to create a 4-block file named SYSTEM.SWAPDISK on the same diskette that contains the Compiler. This allows a segment of the compiler to be swapped out onto the diskette before the operating system segment that opens files is swapped in.

With include files, it also helps to have the (*$I filename*) option in the declaration section of a procedure (before the BEGIN of the

procedure body). This only helps if the compiler swapping option is on and there is a SYSTEM.SWAPDISK file. In the declaration section of a procedure is where the largest section of the compiler can be swapped out to make room for the operating system segment that opens files.

FILE SPACE FOR COMPILER

In Version 1.1, when the code file is automatically sent to the workfile the default size for the file is [*]. In all other cases, the default size is [Ø], which means that the code file will be allocated all of the largest space available on the diskette that it is sent to. If there is only one available space on the diskette, the code file takes all of it.

This can cause problems if the code file is on the same diskette used for the listing file. The Compiler will fail if it tries to create a listing file and the code file has taken all the available space on the specified diskette.

If you run into these problems, specify a different diskette for the code file or the listing file, or specify a definite length for the code file that will leave enough room for other required files.

PROGRAM SEGMENTATION

The information in this section is not needed for simple programs, but can be crucial for programs that are large or complex. This section supplements the information in Chapters 4 and 5 of the Apple Pascal Language Reference Manual. It also describes a new Compiler option, the "Next Segment" option, which is not described in the manual.

To make the most efficient use of the memory space available for program code and data, programs can be divided into segments. This section gives essential information on how the Pascal System implements segmentation.

SEGMENTS

A segment is code that can be loaded into memory and executed, without any other code necessarily being in memory at the same time. Every program consists of at least one segment, and some programs consist of many segments. Whenever a program is compiled, the Compiler and Linker create the following segments in the code file:

- Each SEGMENT procedure or function becomes a segment in the code file.
- Each Regular UNIT that the program uses becomes a segment in the code file.
- The program itself becomes a segment in the code file. This includes the program's non-SEGMENT procedures and functions.

Similarly, whenever a Regular UNIT is compiled, the result is a code segment for the UNIT itself, plus an additional segment for each Regular UNIT that is used within the UNIT being compiled. (Note that SEGMENT procedures and functions are not allowed inside UNITs.)

When an Intrinsic UNIT is compiled, it produces a code segment, and may produce a data segment as well. (Note that an Intrinsic UNIT cannot USE a Regular UNIT.)

Note that segments do not nest -- every segment is just one segment and does not contain any other segments. For example, if a SEGMENT procedure contains another SEGMENT procedure, the result is two distinct code segments.

THE SEGMENT DICTIONARY

Every code file (including library files) contains information called a segment dictionary. This contains an entry for each segment in the code file; the entry has all the information the system needs to load and execute the segment.

The segment dictionary has slots for 16 entries. Therefore, one code file can contain at most 16 segments. In the case of a program, this implies one segment for the program itself, one for each SEGMENT procedure or function, and one for each Regular UNIT used by the program.

Note that Intrinsic UNITs used by a program do not require entries in the segment dictionary of the program's code file. This is because an Intrinsic UNIT's code segment is never in the program's code file -- it is in a library file, and appears in the library file's segment dictionary.

Therefore a program can have a maximum of 16 segments, not counting segments from Intrinsic UNITs. Counting segments from Intrinsic UNITs, a program can have up to 26 segments as explained below.

THE RUN-TIME SEGMENT TABLE

When a program is executed, the Pascal Interpreter uses a segment table which contains an entry for each segment that is used in executing the program. This table thus contains the following entries:

- Entries for 6 segments that the system uses when executing a user program
- An entry for each segment in the segment dictionary of the program's code file
- An entry for each Intrinsic UNIT segment (both data and code segments).

The segment table has slots for up to 32 entries. Since the system uses 6, this means that a program can have up to 26 segments altogether. Remember that only 16 can be in the program's code file; any excess over 16 must be Intrinsic UNIT segments.

SEGMENT NUMBERS

Every segment has a segment number in the range Ø..31. At run time, no two segments in the segment table can have the same number, since the numbers are used to index the table. A segment number is assigned to a program segment when the segment's entry is placed in the code file's segment dictionary (before run time). Numbers are assigned as follows:

- The program itself is Segment 1.
- The segments used by the system are Ø and 2..6.
- The segment number of an Intrinsic UNIT segment is determined by the UNIT's heading, when the Intrinsic UNIT is compiled. (These numbers can be found by examining the segment dictionary of the SYSTEM.LIBRARY file with the LIBMAP utility program.)
- The segment numbers of Regular UNIT segments and of SEGMENT procedures and functions are automatically assigned by the system; they begin at 7 and ascend. Note that after a Regular UNIT is linked into a program, it may not have the same segment number shown for it in the library's segment dictionary when the library is examined with LIBMAP.

To summarize the above, the segment numbers of the program itself, the segments used by the system, and any Intrinsic UNITs used by the program are fixed before the program is compiled; the segment numbers of Regular UNITs and of SEGMENT procedures and functions are not fixed, and are assigned as the program is compiled and linked, in ascending sequence beginning with 7.

Normally, the only time you need to specify segment numbers is in writing an Intrinsic UNIT. You should choose segment numbers that will not conflict with any of the fixed numbers Ø..6 or with any other Intrinsic UNIT that might be used in the same program as the UNIT you are writing.

Intrinsic UNIT segment numbers should also avoid conflict with numbers that might be assigned automatically to Regular UNITs and SEGMENT procedures. However, when unavoidable conflicts arise there is a solution: the new version of the Compiler has a "Next Segment" option which can specify the next automatically assigned segment number. This is explained below.

THE "NEXT SEGMENT" OPTION

This is a new Compiler option which allows you to specify the segment number of the next Regular UNIT, SEGMENT procedure, or SEGMENT function encountered by the Compiler. By default, the segment number is assigned automatically as described above.

(*$NS num*) Sets the next segment number to num, where num is an integer in the range 1..3Ø.

The NS option is ignored if it precedes the program heading; this means that it cannot be used to specify the segment number of the program itself.

The NS option will only work if the specified number is greater than the "default" number that would be automatically assigned. If the number specified in the NS option is less than or equal to the default segment number, the option is ignored. Also, the NS option will not work if the specified number is greater than 3Ø (segment number 31 can only be used for an Intrinsic UNIT).

For example, suppose that you want to use an Intrinsic UNIT named ZEBRA, whose code segment number is 7 and whose data segment number is 8. (Normally, such numbers should be avoided in writing Intrinsic UNITs.) Your program also contains a SEGMENT procedure:

```
PROGRAM ELEPHANT;
USES ZEBRA;
...
SEGMENT PROCEDURE HORSE;
...
```

The Compiler will automatically compile the HORSE procedure as segment number 7, and when you try to execute the program the Interpreter will halt with an error message because the program has two different segments with the number 7. There are two remedies: recompile ZEBRA with different segment numbers (if you have the source for ZEBRA) or use the NS option in your program:

```
PROGRAM ELEPHANT;
USES ZEBRA;
...
(*$NS 9*)
SEGMENT PROCEDURE HORSE;
...
```

Now HORSE will become segment 9 instead of segment 7, and the conflict is avoided.

LOADING OF SEGMENT PROCEDURES AND FUNCTIONS

Normally, the code of a SEGMENT procedure or function is in memory only while it is active; that is, it is loaded from diskette each time the procedure or function is called, and unloaded as soon as it finishes executing. The following program illustrates this:

```
PROGRAM ONE; (*Segment ONE is always in memory.*)

  SEGMENT PROCEDURE ALPHA; (*In memory only when active.*)
    BEGIN
      ...
    END;

  SEGMENT PROCEDURE BRAVO; (*In memory only when active.*)
    SEGMENT PROCEDURE CHARLIE; (*In memory only when active.*)
      BEGIN (*Body of CHARLIE*)
        ...
        ALPHA; (*When this is executed, the segments in
                 memory are ONE, ALPHA, BRAVO, and CHARLIE.*)
        ...
      END;
    BEGIN (*Body of BRAVO*)
      ...
      CHARLIE; (*When this starts executing, the segments in
                 memory are ONE, BRAVO, and CHARLIE.*)
      ALPHA; (*When this is executed, the segments in
               memory are ONE, BRAVO, and ALPHA.*)
      ...
    END;

  BEGIN (*Body of ONE*)
    ...
    ALPHA; (*When this is executed, the segments in
             memory are ONE and ALPHA.*)
    BRAVO; (*When this starts executing, the segments in
             memory are ONE and BRAVO.*)
    ...
  END.
```

The "Resident" option can be used to alter this, as explained below.

LOADING OF UNIT SEGMENTS

Normally, all segments of UNITs used by a program are loaded automatically before the program begins executing, and remain in memory throughout program execution. For example, consider the following program where DELTA and GAMMA are two UNITs, either Regular or Intrinsic:

```
PROGRAM TWO
USES DELTA, GAMMA;
...
BEGIN
  ...
END.
```

Throughout program execution, the segments in memory are TWO, DELTA, and GAMMA. This can be altered by the "Noload" option, as explained below.

THE "NOLOAD" OPTION

The "Noload" option, (*$N+*), is described in Chapter 4 of the Apple Pascal Language Reference Manual; this section explains its usage.

The (*$N+*) option is placed at the beginning of the main program body (after the BEGIN). It causes all UNIT segments to be handled in the same way as SEGMENT procedures, during program execution. With this option a UNIT segment is in memory only when something in its INTERFACE part is referenced by the program.

The (*$N+*) option does not prevent the initialization part of a UNIT from being loaded and executed before program execution; but after initialization the UNIT segment is unloaded until it is activated.

Consider the following program, where HUGEPROC is a large procedure and BIGUNIT is a large UNIT. The system does not have enough memory to hold HUGEPROC and BIGUNIT at the same time, along with the program itself.

```
PROGRAM THREE;
USES BIGUNIT;

SEGMENT PROCEDURE HUGEPROC;
  BEGIN
    ...
  END;

BEGIN
  (*$N+*) (*Keeps BIGUNIT out of memory until needed.*)
  HUGEPROC;
  ...
  CALCULATE; (*A procedure in BIGUNIT*)
  ...
  HUGEPROC
END.
```

First HUGEPROC is called; BIGUNIT is not in memory because of the (*$N+*) option. When CALCULATE is called, HUGEPROC is not in memory since it is a SEGMENT procedure. As soon as no part of BIGUNIT is active, it is again swapped out of memory, and HUGEPROC can be called again.

THE "RESIDENT" OPTION

The "Resident" option is described in Chapter 4 of the Apple Pascal Language Reference Manual; this section explains its usage.

The "Resident" option is placed at the beginning of the body of a procedure or function (after the BEGIN). It alters the handling of segments that would otherwise be in memory only when active: that is, SEGMENT procedures and functions, and UNITs under the "Noload" option. When such a segment is named in the "Resident" option, it is immediately loaded into memory and remains there as long as the procedure or function containing the "Resident" option is active. For example, consider the following program:

```
PROGRAM FOUR;
USES BIGUNIT;

SEGMENT PROCEDURE HUGEPROC;
  BEGIN
    ...
  END;

PROCEDURE CALLHUGEPROC;
  VAR I: INTEGER;
  BEGIN
    FOR I:=1 TO 100 DO HUGEPROC
  END;

PROCEDURE CALLCALCULATE;
  VAR I: INTEGER;
  BEGIN
    FOR I:=1 TO 100 DO CALCULATE (*A procedure in BIGUNIT*)
  END;

BEGIN
  (*$N+*) (*Keeps BIGUNIT out of memory until needed.*)
  HUGEPROC;
  ...
  CALCULATE;
  ...
  CALLHUGEPROC;
  ...
  CALLCALCULATE
END.
```

This resembles the previous example, but the CALLHUGEPROC and CALLCALCULATE procedures are new. As written, these two procedures have a problem: since HUGEPROC is a SEGMENT procedure, it will be swapped in from diskette 1ØØ times when CALLHUGEPROC executes, and because of the (*$N+*) option in the main program body, BIGUNIT will be swapped in 1ØØ times when CALLCALCULATE executes. This is obviously undesirable, and it can be prevented by using the "Resident" option in each of these procedures:

```
PROCEDURE CALLHUGEPROC;
  VAR I: INTEGER;
  BEGIN
  (*$R HUGEPROC*)
    FOR I:=1 TO 1ØØ DO HUGEPROC
  END;

PROCEDURE CALLCALCULATE;
  VAR I: INTEGER;
  BEGIN
  (*$R BIGUNIT*)
    FOR I:=1 TO 1ØØ DO CALCULATE (*A procedure in BIGUNIT*)
  END;
```

Now HUGEPROC will be kept in memory as long as CALLHUGEPROC is active, and BIGUNIT will be kept in memory as long as CALLCALCULATE is active.

Finally, note that the "Resident" option can be applied to more than one segment, by separating the names of segments with commas as in the following example:

```
(*$R ALPHA,BETA,GAMMA*)
```

where ALPHA, BETA, and GAMMA are names of segments (UNITs, SEGMENT procedures, or SEGMENT functions). The option shown would make all three segments resident in the procedure containing the option.