Apple II Technical Notes

Developer Technical Support

Pascal #10: Configuration and Use of the Apple II Pascal Run-Time Systems

Revised by: Cheryl Ewy November 1988
Revised by: Cheryl Ewy June 1985

This Technical Note describes the Apple II Pascal Run-Time Systems which permit the “turnkey” execution of application software which has been developed using Apple Pascal.

System Overview

The Run-Time Systems support only the execution of an application package. Unlike the Pascal Development System, the Run-Time Systems do not contain the Assembler, Compiler, Editor, Filer or Linker, nor even an error reporting mechanism at the system level. System operations such as transferring files, compacting disks (Krunching), and the reporting of and recovery from errors, are all left to the application program. It is the software developer’s responsibility to design and implement friendly, entirely self-contained packages for use with the Run-Time Systems. The safest assumption to make when developing such packages is that the user is not only unfamiliar with the facilities of the Pascal Development System, but may also be ignorant of computer operation and use in general.

The three run-time systems currently available are :

- The 48K Run-Time System V1.2 (standard and stripped)
- The 64K Run-Time System V1.3 (standard only)
- The 128K Run-Time System V1.3 (standard only)

The name of each Run-Time System indicates the minimum amount of RAM necessary for proper operation. Any additional RAM available will not be used by the Run-Time Systems.

The 48K Run-Time System has **not** been updated to version 1.3, as have the 64K and 128K Run-Time Systems. Thus, the changes and improvements made to Pascal for version 1.3 are not available in the 48K Run-Time System. Specifically, the 48K Run-Time System can only use Disk II drives and can only boot from slot 6. See the *Apple II Pascal 1.3 Manual* for more information on the differences between versions 1.2 and 1.3 of Apple II Pascal.

There are two configurations of the 48K Run-Time System available, one of which provides more free memory for the application package's programs and data than does the other. Except as noted later, the standard configuration of the Run-Time System supports all features of the Pascal Development System that are relevant to turnkey execution of application software. The stripped configuration lacks set operations and floating-point arithmetic.

Contents of the Apple II Pascal Run-Time System Disks

The following files are contained on the Apple II Pascal 1.2 48K Run-Time System disk (RT48:):

- RTSTND.APPLE 48K Run-time standard P-machine.
- RTSTRP.APPLE 48K Run-time stripped P-machine.
- SYSTEM.PASCAL 48K Run-time operating system.
- RTBSTND.BOOT Contains the boot code for RTSTND.APPLE.
- RTBSTRP.BOOT Contains the boot code for RTSTRP.APPLE.
- RTBOOTLOAD.CODE Utility program to load 48K Run-time boot code onto blocks 0 and 1 of Vendor Product disk.

The following files are described below:

- SYSTEM.LIBRARY
- SYSTEM.ATTACH
- RTSETMODE.CODE
- II40.MISCINFO
- II80.MISCINFO
- IIE40.MISCINFO
- SYSTEM.MISCINFO
- SYSTEM.CHARSET

The following files are contained on the Apple II Pascal 1.3 64K Run-Time System disk (RT64:):

- SYSTEM.APPLE 64K Run-time standard P-machine.
- SYSTEM.PASCAL 64K Run-time operating system.

The following files are described below:

- SYSTEM.LIBRARY
- SYSTEM.ATTACH
- RTSETMODE.CODE
- II40.MISCINFO
- II80.MISCINFO

- SYSTEM.MISCINFO
- SYSTEM.CHARSET

The following files are contained on the Apple II Pascal 1.3 128K Run-Time System disk (RT128:):

- SYSTEM.APPLE 128K Run-time standard P-machine.
- SYSTEM.PASCAL 128K Run-time operating system.

The following files are described below, and are identical to the 64K Run-Time System files:

- SYSTEM.LIBRARY
- SYSTEM.ATTACH
- RTSETMODE.CODE
- SYSTEM.MISCINFO
- SYSTEM.CHARSET

The Development Systems referred to in the following file descriptions are the Apple II Pascal 1.3 Development System when discussing files on the 64K and the 128K Run-Time System disks and the Apple II Pascal 1.2 Development System when discussing files on the 48K Run-Time System disk.

SYSTEM.LIBRARY	contains the run-time versions of the same Intrinsic Units supplied with the Development System. These Units are for use only with the Run-Time System and will not execute properly in the Development environment. Conversely, only the Units in this library, **not** those on the Development System disks, should be used when executing programs in the Run-time environment. Note that the developer is free to add his own Intrinsic Units to SYSTEM.LIBRARY.
SYSTEM.ATTACH	is a run-time version of the dynamic driver-attachment program described in *Apple II Pascal Device and Interrupt Support Tools*. This version may only be used with the Run-Time Systems.
RTSETMODE.CODE	is a utility program that permits the vendor to arm or disarm any or all of four system options: Filehandler Overlay, Single Drive System, Ignore External Terminal, and Get/Put and Filehandler Overlay.
MISCINFO	files are identical to those supplied on the Development System disks and are supplied here only for the sake of redundancy.
SYSTEM.CHARSET	is identical to the file supplied with the Development System; it is included here only for the sake of redundancy. This file is needed on the Vendor Product Disk only if TURTLEGRAPHICS is used.

Of the files supplied on the Run-Time System disks, the final Vendor Product Disk should contain only the Run-time P-machine (SYSTEM.APPLE, RTSTND.APPLE, or RTSTRP.APPLE), SYSTEM.PASCAL, SYSTEM.LIBRARY, the appropriate MISCINFO file renamed to SYSTEM.MISCINFO, and, optionally, SYSTEM.CHARSET. SYSTEM.ATTACH, with its attendant data files should be included on the Vendor Product Disk if special device drivers must be bound into the system for use by the Application Package. All other files on the Run-Time System disks are used in creating and configuring the Vendor Product Disk.

Operation

The term Vendor Product Disk, as used throughout this Technical Note, refers to the primary (boot) disk in a turnkey application package, which is assumed to contain the following software: the Run-time P-machine, the Run-time Operating system, a SYSTEM.LIBRARY file, a SYSTEM.MISCINFO file, and the files comprising the application package's programs (and any necessary data). In most instances, the Vendor Product Disk will be the only software disk in the package. Larger systems, however, may also include other disks that contain additional software and data which will not fit on the boot disk.

Note that the main application program must be named SYSTEM.STARTUP, so the Run-Time System can find it when booting.

A two-stage boot process can be used with the 64K and 128K Run-Time Systems if the necessary boot files listed above cannot fit on a single disk. In this case, the primary boot disk would contain only the Run-time P-machine. A second-stage boot disk would contain the remainder of the files. A two-stage boot process cannot be used with the 48K Run-Time System.

The Boot Process

The boot code (contained in blocks 0 and 1 of the boot disk) is loaded into memory by the Autostart ROM. It checks for the P-machine file and loads it into RAM. The P-machine, in turn, brings in and initializes the Run-time operating system. (In the case of a two-stage boot, the message "Insert boot disk with SYSTEM.PASCAL on it, then press RETURN" appears after the P-machine has been loaded. The user should then insert the second-stage boot disk and press the Return key, which results in the Run-time operating system being loaded and initialized.) The first noteworthy action taken by the operating system is to execute SYSTEM.ATTACH, if that utility program is available on the Vendor Product Disk. Remember that SYSTEM.ATTACH must not be present on the Vendor Product Disk unless special, low-level I/O drivers must be bound into the system. As explained more fully in *Apple II Pascal Device and Interrupt Support Tools*, SYSTEM.ATTACH uses two special data files and will fail if these files are not present on the boot disk. Putting SYSTEM.ATTACH on the Vendor Product Disk without also providing the required data files insures consistent failure of the system boot process. It is possible to include SYSTEM.ATTACH on the Vendor Product Disk, while defeating the automatic execution of it at boot time, by changing its name.

The boot process culminates when the main application program, SYSTEM.STARTUP, is loaded and executed. Any failure during the boot process is fatal. Whenever possible, a failure will display the following message:

```
SYSTEM FAILURE NUMBER nn.  PLEASE REFER TO PRODUCT MANUAL.
```

Here, nn refers to the actual number reported when the failure occurs. This number corresponds to one of the following failures:

01 Unable to load specified program
02 Specified program file not available
03 Specified program file is not code file
04 Unable to read block zero of specified file
05 Specified code file is un-linked
06 Conflict between user and intrinsic segments
07 UNASSIGNED ERROR CODE
08 Required intrinsics not available
09 System internal inconsistency
10 Can't load required intrinsics/Can't open library file
11 Specified code file must be run under the 128K system
12 Original disk not in boot drive

Clearly, these messages are useful as debugging tools as well as in mechanisms for field failure reporting. The Product Manual mentioned in the bootstrap failure message is, of course, the

vendor's own product manual. It is the responsibility of the vendor to enumerate and explain for the user the situations in which bootstrap failures may occur, as well as suggest remedies for these failures.

General Considerations

Once the program is loaded and running, operation proceeds normally and may even include removal of the system disk. (It is, however, the responsibility of the application package to protect itself against the possibility that the system disk will not be on-line when a segment must be loaded or when a specific subprogram must be chained to. At such times, the application software should first determine whether or not the required disk is on-line, and, if not, suspend operation, after

giving a suitable prompt, until the user has inserted the disk in the appropriate drive.) Any errors that occur during execution of the application package cause the system to transfer program control to a specific procedure in the currently-executing application program, where code intended to respond to errors is assumed to exist. If any program in the application system terminates without chaining to another one, the Run-time system reboots into SYSTEM.STARTUP.

Specifications

Available Configurations

The memory requirements of different applications impose the need for different Run-Time Systems. The developer should choose one of the systems as the target environment, and keep its limitations and capabilities in mind during design and implementation of the application package. Apple currently supports the following Run-Time Systems:

- The 48K Run-Time System V1.2 (standard and stripped)
- The 64K Run-Time System V1.3 (standard only)
- The 128K Run-Time System V1.3 (standard only)

The difference between the standard and stripped versions of the 48K Run-Time System is that the stripped version does not support set operations or floating point arithmetic, thereby making more memory available for the application.

The chart below summarizes the amount of free memory that is available under the different Run-Time Systems for use by the application package. Note that when swapping is set to level 1, the amount of memory available to the application package is increased by approximately 3660 bytes.

	No Swapping	Swapping on level one
48K Standard	23372 Bytes	27040 Bytes
48K Stripped	25676 Bytes	29344 Bytes
64K	40290 Bytes	43958 Bytes
128K (code)	40758 Bytes	44410 Bytes
128K (data)	44502 Bytes	44526 Bytes

Figure 1–Free Memory in Run-Time Systems

Note: The amount of free memory available with the 64K Run-Time System is reduced by 1024 bytes if it is operating in 40-column mode. Similarly, the amount of free

memory available for data in the 128K Run-Time System is reduced by 1024 bytes if the system is operating in 40-column mode.

There is another level of swapping (level 2) which provides an additional 810 bytes of usable memory, however, using `GET` or `PUT` to disk will be slow if swapping level 2 is selected since these routines will have to be loaded from disk repeatedly. `READ` and `WRITE` to disk will also be slow since they use `GET` and `PUT`. `BLOCKREAD`, `BLOCKWRITE`, `UNITREAD`, and `UNITWRITE` will be unaffected.

Swapping can be set to the desired level by using `RTSETMODE` (described later) or by calling a procedure in `CHAINSTUFF` before chaining to another subprogram. See the *Apple II Pascal 1.3 Manual* for further information on swapping.

Use Environment

The hardware environment must include the following:

48K Run-Time System	An Apple][or][+ with 48K of RAM (minimum), or an Apple IIe, IIc or IIGS.
64K Run-Time System	An Apple][or][+ with 48K of RAM and an Apple Language Card, or an Apple IIe, IIc or IIGS.
128K Run-Time System	An Apple IIe with an Extended 80-Column Text Card, an Apple IIc or an Apple IIGS.
All Run-Time Systems	At least one disk drive in slot 4, 5, or 6. Video screen or external terminal (video screen preferred).

Note that the Run-Time Systems support all standard Apple peripheral cards. Other cards may not operate properly, especially if they include firmware that depends upon specific internal characteristics of the P-machine or operating system. SYSTEM.ATTACH must be used by those vendors who wish to reconfigure the BIOS (Basic I/O Subsystem) to support non-standard peripheral devices. Through the `ATTACH` facility, it is possible to assign new physical devices to any of the existing logical I/O units in the Pascal system, as well as retain the standard device assignments while adding new devices to the system. Drivers prepared for use with SYSTEM.ATTACH are bound into the system dynamically when it boots. Note that the addition of special I/O drivers to the system will reduce the amount of free memory available for use by the applications code, since drivers are loaded on the Pascal system heap. For more information, see *Apple II Pascal Device and Interrupt Support Tools*.

Restrictions and Considerations

1. SYSTEM.ATTACH and the CHAINSTUFF, LONGINTIO, and PASCALIO units in SYSTEM.LIBRARY make assumptions about the internal structure of the Pascal operating system. Because the internals of the Run-time operating systems are different from those in the Development System, only the versions of CHAINSTUFF, LONGINTIO, PASCALIO and SYSTEM.ATTACH that are supplied on the Run-Time System disks should be used in the Run-time execution environment. (These special versions should never be used in the Development environment.)

2. The units TRANSCEND and TURTLEGRAPHICS employ floating-point operations, so software intended to be executed under the 48K stripped Run-Time

System should not use them. For software that employs the TURTLEGRAPHICS procedure `TURNTO`, note that turns through right angles and null angles are treated as special cases, and the TURTLEGRAPHICS unit uses only integer arithmetic in calculating the trigonometric values needed to execute them. TURTLEGRAPHICS may be used under the 48K stripped Run-Time System if the turtle is allowed to make only right-angle turns (i.e., the HILBERT demonstration program on the APPLE3: disk). Attempts to draw arbitrary curves, as demonstrated in the GRAFDEMO program on APPLE3:, will produce execution errors in the 48K stripped Run-time environment.

3. Pascal's special function keys retain their meanings in the Run-Time Systems. The following keys have special meanings:

Control-@	Break
Control-A	Switch to alternate half of screen
Control-F	Flush screen display
Control-S	Freeze (Stop) screen display
Control-Z	Initiate auto-follow mode
Control-W, Control-E	Upper/lower case activation
Control-R, Control-T	Reverse video toggles
Control-K	Left square bracket
Shift-M	Right square bracket

   Note: Some of these special function keys are ignored by Pascal if it is running on an Apple IIe, IIc or IIGS. Also, it is possible to disable some of these special key functions. See *Apple II Pascal 1.3 Manual* for complete details.

4. The Run-Time System will operate correctly only with programs that have been prepared for execution in the Apple II Pascal environment.using Apple's Pascal compiler or Pascal-system assembler on either an Apple II or an Apple ///.

5. The Run-Time System is optimized for operation with Apple's built-in video output screen. There is no easy way for a turnkey package to reconfigure its host Run-Time System to use the random-cursor facilities of any arbitrary external terminal. Therefore, it is expected that users of the system will be operating with the standard Apple video screen and not an external terminal. Any program that makes use of screen control, such as clearing the screen, random cursor addressing, or backspacing, is not likely to work properly on an external terminal. To avoid this problem, the Run-Time System contains a switch which can be set through the RTSETMODE program (explained below). When set, this switch causes the system to ignore an external terminal, if one is connected. Simple programs that do not make use of any screen control may leave the external terminal switched in without any adverse consequences.

Run-Time System Configuration Utilities

RTSETMODE (provided with all Run-Time Systems)

Flags which note the state of four system options are contained within a special part of the directory of any Run-Time System boot disk. (These flags will not normally be present on disks prepared for or used with the Pascal Development System.) When a flag is set (TRUE),

the corresponding system option is enabled. The option is disabled when the corresponding flag is reset (FALSE). At boot time, the option flags are checked and are used during a dynamic configuration process which occurs before the application software is executed.

The RTSETMODE utility is used by the application developer to set or reset the option flags, according to the requirements of the application package. In operating RTSETMODE, the developer first selects the Pascal volume to be affected, then answers four yes-or-no questions by pressing the Y or N keys, respectively. Responding to any prompt for input by pressing only the Return key causes immediate termination of the program.

Answering yes to any of the following questions **arms** the indicated option (setting the corresponding flag), while answering no **disarms** the option (and resets the corresponding flag).

> Arm Filehandler Overlay Option? Arming this option sets OS swapping to level 1. Operating System code related to disk file opening and closing is swapped into memory as needed by the application software, thus freeing approximately 3660 bytes of RAM for use by the application.
>
> Arm Single-drive System Option? With this option armed, the initial boot process is finished, the Pascal system will not assume the availability of any disk drives other than the boot drive. Specifically, volume searches will be limited to the boot drive. The application may still use Apple Pascal's `UNITREAD` and `UNITWRITE` procedures to access any other drives which may be connected to the system.
>
> Arm Ignore External Terminal Option? Arming this option insures that the Pascal system will always operate in 40-column mode, regardless of whether or not an external terminal interface or 80-column card is available.
>
> Arm Get/Put and Filehandler Overlay Option? Arming this option sets OS swapping to level 2. Operating System code related to disk file opening and closing, as well as `GET` and `PUT` to disk is swapped into memory as needed. (See above for more information on swapping level 2.)

After the four-question sequence, RTSETMODE asks the user to confirm that all information input to that point is correct and should be used to update the Vendor Product Disk. If so, an attempt is made to update the disk's directory with the new set of option flags, and RTSETMODE finishes by reporting the success or failure of the update operation.

Developers should note that only exact copies of a Run-time boot disk will retain its option flags. Transferring the Run-Time System and applications software from disk to disk on a file-by-file basis will not transfer the option flags between the disks. For this reason, it is recommended that RTSETMODE be applied to the product master of any package based on Run-time immediately prior to releasing that master to production, to insure the correct status of the option flags.

If a two-stage boot will be used for a run-time application, RTSETMODE must be run on both boot disks since the flags are checked by both the P-machine and the operating system.

RTBOOTLOAD (48K Run-Time System only)

This program is used to transfer to the Vendor Product Disk the proper boot code for the chosen 48K Run-time configuration (STND or STRP). Responding to any prompt for input by

pressing only the Return key results in immediate termination of the program. RTBOOTLOAD first asks for the name of the file which contains the appropriate boot code (either RTBSTND.BOOT or RTBSTRP.BOOT). The filename must be entered exactly as it appears in the directory (including a volume prefix if the file is not on the default volume), or the program will not be able to find the file, and will repeat its request for a filename. Once it has fetched the boot code, RTBOOTLOAD asks for the volume name of the Vendor Product Disk, then waits for the user to press the space bar (thus providing the user with an opportunity to insert the selected volume, if necessary) before attempting to transfer the boot information. The success or failure of the transfer is reported before RTBOOTLOAD terminates. This program is only supplied on the 48K Run-Time System disk and should never be used to transfer boot information to a disk which contains the 64K or 128K Run-Time Systems, as doing so will prevent the systems from booting correctly.

Error Handling

If an error in execution or I/O occurs during program operation, the Run-Time System attempts to let the application package itself acknowledge, and if possible, recover from the error condition. As with the Pascal Development environment, the application developer is free to use the `$I-` and `$R-` compiler options to assume localized, programmatic control of the corresponding error situations.

When the Run-Time System detects an error, it stores the error number in `IORESULT` and calls `PROCEDURE NUMBER TWO` of the currently-executing program. This is the procedure in segment number 1 that has been given the procedure number 2 by the compiler. In other words, it is the first one declared after the program heading that is not itself a unit or segment procedure, or within a unit or segment procedure. In a compiler listing, `PROCEDURE NUMBER TWO` may be identified as those lines whose S (segment) number is 1, and whose P (procedure) number is 2.

`PROCEDURE NUMBER TWO` may be declared as a forward procedure since the procedure number is assigned at the forward declaration.

From now on, `PROCEDURE NUMBER TWO` will usually be called the error handler, since it must always be reserved by the application programmer for the sole purpose of handling errors. The error handler may not have any parameters, and must always be declared as a `PROCEDURE`, never as a `FUNCTION`.

The error handler can determine what kind of error has occurred by checking the value of the `IORESULT` function. In the Development System, this function is restricted to containing the codes for any I/O errors that might occur during execution. In the Run-Time Systems, `IORESULT` has been extended to report all system errors, as well as the usual I/O errors.

Here are all the values `IORESULT` can assume during Run-time execution:

00 No error	100 Unknown Run-time error
01 Bad block, parity error	101 Value range error
02 Illegal unit number	102 No procedure in segment table
03 Illegal I/O request	103 Exit from uncalled procedure
04 Data-com timeout	104 Stack overflow
05 Volume went off-line	105 Integer overflow
06 File lost in directory	106 Divide by zero
07 Bad file name	107 Nil pointer reference
08 No room on volume	108 Program interrupted by user
09 Volume not found	109 System I/O error
10 File not found	110 User I/O error

11 Duplicate directory entry
12 File already open
13 File not open
14 Bad input format
16 Disk is write-protected
17 Illegal block number
18 Illegal buffer address
19 Must read a multiple of 512 bytes
20 Unknown ProFile error
64 Device error
111 Unimplemented instruction
112 Floating point error
113 String overflow
114 Programmed HALT
115 Programmed breakpoint
116 Codespace overflow

It is recommended that a program's error handler should simply report system error for all cases except those which are relevant to the program. Global state variables in the program may be used to help determine the nature of the problem and report it to the user. Note that a system reboot occurs if an attempt is made to exit the program (without chaining to another).

After the error handler finishes its operation, control returns to the caller of the procedure where the error occurred (unless the error was fatal). In this way, program operation may be continued, cleanly and simply, after an error is handled. The caller of a failure-prone procedure can set and test status flags to determine whether or not the called procedure completed its operation and either repeat the procedure call or perform an alternative action.

In developing particularly large systems where program chaining is used, the application programmer should remember that each chained program must reserve `PROCEDURE NUMBER TWO` as an error handler.

Following are two programming examples. The first shows a typical error handler routine, and the second is a program fragment that demonstrates an error recovery technique.

```
(* EXAMPLE #1 — ERROR HANDLER *)

       (* THE FOLLOWING PROCEDURE IS ONLY *)
       (* CALLED BY THE OPERATING SYSTEM *)

       PROCEDURE ErrorHandler;

              PROCEDURE Message(Space: Boolean; S: String);
              VAR Ch : Char;
              BEGIN (* Message *)
                     WriteLn;
                     WriteLn('*** ',S);
                     IF Space THEN
                            BEGIN
                                   Write('*** Press SPACE-BAR to continue');
                            REPEAT
                                          Read(Keyboard, Ch)
                            UNTIL ((Ch = ' ') AND (NOT EoLn));
                            END;
                     END   (* Message *);

              BEGIN (* ErrorHandler *)
                     IF (IOResult = 14) THEN
                            Message(True,'That is not a legal integer!')
                     ELSE IF (IOResult = 106) THEN
                            Message(True,'Division by zero is impossible!')
                     ELSE BEGIN
                            Message(False,'System error.  Please reboot.');
                            WHILE True DO (* Hang *);
                     END;
              END   (* ErrorHandler *);

              (* END OF EXAMPLE #1 *)
```

```
(* EXAMPLE #2 — ERROR RECOVERY USING ERROR HANDLER OF EXAMPLE #1 *)

PROCEDURE Calculator;
(* Features recovery from input or arithmetic error. *)
 TYPE Order = (First, Second);
VAR A,B : Integer;
        Flag : Boolean;

       PROCEDURE GetNumber(Which: Order; VAR Number: Integer);
       BEGIN
              Write('Input the');
              IF (Which = First) THEN
                     Write(' first')
              ELSE Write(' second');
              Write(' number: ');
               Read(Number);  ReadLn;
               Flag := True;
       END   (* GetNumber *);

        PROCEDURE Answer;
              VAR R : Real;
              BEGIN
                     R := A / B; (* Bombs if B=0 *)
                     WriteLn;
                     WriteLn(A,' divided by ',B,' is ',R);
               END   (* Answer *);

       BEGIN (* Calculator *)
              REPEAT
                     Flag := False;
                     WriteLn;
                     WriteLn;
                     REPEAT
                            GetNumber(First,A)
                     UNTIL Flag;
                     Flag := False;
                     WriteLn;
                     REPEAT
                            GetNumber(Second,B)
                     UNTIL Flag;
                     Answer;
              UNTIL Eof;
       END   (* Calculator *);

       (* END EXAMPLE #2 *)
```

To illustrate the effect of the Run-Time System's error handling mechanism, here is the interaction between user and machine during a typical run of the above Calculator program. User-input is terminated by a press of the Return key in all cases except the first and last. In the first case, the error handler is invoked during the erroneous numeric input. In the last case, the system accepts and acts upon a Control-C signal before the user has a chance to press any other keys.

```
Input the first number: N

*** That is not a legal integer!

Input the first number: 16

Input the second number: 0

*** Division by zero is impossible!

Input the first number: 16

Input the second number: 2

16 divided by 2 is 8

Input the first number: <Control-C>
```

As soon as the user presses Control-C, the Run-time system detects the end of the standard input file (EOF), and reboots (right back into Calculator).

Differences between the Pascal Development Systems and the Run-Time Systems

Although the Run-Time Systems will run most Pascal code files exactly as does the Pascal Development System, the application developer must be aware of important differences between the two environments. As mentioned above, there is no system-level handling of any type of error that may occur, including stack overflow, arithmetic errors, or bad disk reads. It is left to the application package to respond to all error conditions. The typical user will not have access to (nor knowledge of) the Pascal Formatter or Filer.

Many programs which fit comfortably in the 64K Development System environment may fail to execute at all under the 48K Run-Time System due to the difference in available user memory. Similarly, programs developed with the 128K Development System may fail to execute under the 64K Run-Time System for the same reason. While large systems can be made to fit within the confines of a particular Run-time environment, this is possible only through use of Apple Pascal's program segmentation (overlay) and chaining facilities. It is suggested, however, that much thought and care be taken when using chaining and segmentation in software design, since these facilities, by their very nature, involve time-consuming disk accesses. Application software that abuses chaining or segmentation, or employs them in a careless fashion, may easily waste a large amount of time in disk thrashing, especially if swapping is being used. Finally, an application package runs the risk of massive failure unless calls to program overlays and chaining are preceded by checks that the expected disk is in the appropriate drive. This is especially important when the target machine includes only one disk drive (as is frequently the case).

The following items are never present in the Run-Time Systems:

- System `HOMECURSOR`, `CLEARSCREEN`, and `CLEARLINE` functions
- System prompt function
- Compiler, Assembler, Linker, Editor, and Filer
- `IDSEARCH` and `TREESEARCH` procedures

Programs that make use of information stored in specific memory locations within the Development System P-machine or that make assumptions about static or dynamic memory allocation at the operating system level (i.e., for the purpose of accessing system data structures) are likely to function incorrectly when executed in the Run-time environment. This failure to run is due to the code reorganization, compaction, and optimization that was necessary to produce the Run-Time Systems.

Creation of Vendor Product Disks

The following steps can be used as a guide for creating a Vendor Product Disk:

1. Format a disk using the Pascal Development System Formatter.
2. Transfer the files SYSTEM.APPLE (or RTSTND.APPLE or RTSTRP.APPLE), SYSTEM.PASCAL, SYSTEM.LIBRARY, SYSTEM.MISCINFO, and SYSTEM.CHARSET (if needed) from the Run-Time System disk to the Vendor Product Disk.
3. Transfer the code file or files for the application to the Vendor Product Disk. The main code file for the application must be named SYSTEM.STARTUP.
4. Run the Pascal Development System Library program to add any needed library units to SYSTEM.LIBRARY on the Vendor Product Disk.
5. Run RTBOOTLOAD to load the appropriate bootstrap code from RT48: onto the Vendor Product Disk. (48K Run-Time Systems Only)
6. Run RTSETMODE if you wish to arm the Filehandler Overlay option, the Single-Drive System option, the Ignore External Terminal option, or the Get/Put and Filehandler Overlay option.

Vendor Product Disks, or other disks which contain 48K Run-Time System software should be copied using only whole-volume transfer mechanisms, such as that provided by the Pascal system Filer. A succession of individual file transfers, or a wildcard transfer (such as transferring #4:= to #5:$), will only copy files from one disk to another. They will not copy the crucial 48K Run-time boot code between disks. Only whole-volume transfers (such as #4: to #5:, or SOUP: to NUTS:) will result in complete copies, containing the proper boot information.

Vendor Product Disks, or other disks which contain 64K or 128K Run-Time System software

can be copied using either whole volume or individual file transfers since they do not contain special bootstrap information.

Apple FORTRAN and the Run-Time Systems

Apple FORTRAN programs will execute correctly under the Apple II Pascal Run-Time Systems (48K and 64K only), as long as no execution errors or untrapped I/O errors occur. Using only FORTRAN, it is impossible to produce object code that contains the specially-placed error-handling procedure to which control is transferred in the event of an untrapped error during Run-time execution. Furthermore, the FORTRAN Run-Time Support Library includes system-level code for handling FORTRAN I/O errors independently of the Apple Pascal system's own error-handling facilities. Execution of this special code will always lead to a system reboot in the Run-time environment.

Users who wish to provide turnkey packages based on FORTRAN object-code are advised to link the FORTRAN object-code to a Pascal host, as explained in the *Apple FORTRAN Language Reference Manual*. The only live code which the Pascal host must contain is the error-handling procedure that the Run-Time Systems require for robust execution of turnkey software.

Further Reference

- *Apple II Pascal 1.3 Manual*
- *Apple II Pascal Device and Interrupt Support Tools*
- *Apple FORTRAN Language Reference Manual*