APPLE COMPUTER, INC.
20525 Mariani Avenue
Cupertino, CA 95014

PASCAL TECHNICAL NOTE #10

Configuration and Use of The Apple II Pascal 1.2 Runtime Systems

(December 1983)

For further information contact:
PCS Developer Technical Support
M/S 22-W, Phone (408) 996-1010

Disclaimer of All Warranties and Liabilities

Apple Computer, Inc. makes no warranties, either express or implied, with respect to this documentation or with respect to the software described in this documentation, its quality, performance, merchantability, or fitness for any particular purpose. Apple Computer, Inc. software is sold or licensed "as is". The entire risk as to its quality and performance is with the vendor. Should the programs prove defective folowing their purchase, the vendor (and not Apple Computer, Inc., its distributor, or its retailer) assumes the entire cost of all necessary servicing, repair, or correction and any incidental or consequential damages. In no event will Apple Computer, Inc. be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the software, even if Apple Computer, Inc. has been advised of the possibility of such damages. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation may not apply to you.

This documentation is copyrighted. All rights are reserved. This document may not, in whole or part, be copied, photocopied, reproduced, translated or reduced to any electronic medium or machine readable form without prior consent, in writing, from Apple Computer, Inc.

Copyright 1983 by Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014
(408) 996-1010

Notice

Apple Computer, Inc. reserves the right to make improvements in the product described in this document at any time and without notice.

APPLE COMPUTER, INC.
20525 Mariani Avenue
Cupertino, CA 95014

PASCAL TECHNICAL NOTE #10

Configuration and Use of The Apple II Pascal 1.2 Runtime Systems

(December 1983)

For further information contact:
PCS Developer Technical Support
M/S 22-W, Phone (408) 996-1010

I. INTRODUCTION

The Apple II Pascal 1.2 Runtime Systems permit the "turnkey" execution of application software that has been developed using Apple Pascal. This Technical Note is intended to aid Vendors and applications developers who are familiar with the Apple II Pascal 1.2 Development System. Those who are not should read carefully the following documents:

* *Apple Pascal Operating System Reference Manual* (with addendum)
* *Apple Pascal Language Reference Manual* (with addendum)
* *Apple II Pascal 1.2 Update Manual*

II. SYSTEM OVERVIEW

The Runtime Systems support only the execution of an application package. Unlike the Pascal Development System, the Runtime 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, disk compacting ("Krunching"), and the reporting of and recovery from errors, are all left to the application program. Clearly, it is the software developer's responsibility to design and implement "friendly," entirely self-contained packages for use with the Runtime Systems. The safest assumption to make when developing such packages is that the end-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 runtime systems currently available are :

* The 48K Runtime System (standard and stripped versions)
* The 64K Runtime System (standard version only)
* The 128K Runtime System (standard version only)

The name of each runtime system indicates the minimum amount of RAM necessary for proper operation. Any additional RAM available above the minimum will not be used by the Runtime Systems.

There are two versions of the 48K Runtime 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 Runtime System supports all features of the Pascal Development System that are relevant to turnkey execution of applications software. The "stripped" configuration lacks set operations and floating-point arithmetic.

III. CONTENTS OF APPLE II PASCAL 1.2 RUNTIME DISKETTES

The following files are contained on "RT48:", the Apple II Pascal 1.2 48K Runtime System diskette:

* RTSTND.APPLE (29 blocks) — 48K Runtime "standard" P-machine.
* RTSTRP.APPLE (24 blocks) — 48K Runtime "stripped" P-machine.
* SYSTEM.PASCAL (28 blocks) — 48K Runtime operating system.
* SYSTEM.LIBRARY (39 blocks) — Contains the same Intrinsic Units as described in the Apple Pascal Language Reference Manual. However, these Units are for use only with the Runtime System, and will not execute properly in the development environment. Conversely, only Units in this library, NOT those on the 1.2 Development System diskettes, should be used when executing programs in the Runtime environment. Note that the developer is, however, free to add his own Intrinsic Units to the Runtime SYSTEM.LIBRARY.
* SYSTEM.ATTACH (9 blocks) — A runtime version of the dynamic driver-attachment program described in the Apple II Pascal Attach Tools manual. This version may only be used with the Runtime Systems.
* RTSETMODE.CODE (4 blocks) — Utility program that permits Vendor to arm or disarm any or all of four configuration options: "Filehandler Overlay", "Single Drive System", "Ignore External Terminal" and "Get/Put and Filehandler Overlay".
* RTBOOTLOAD.CODE (4 blocks) — Utility program to load 48K Runtime bootstrap code onto blocks 0 and 1 of Vendor Product Diskette.
* RTBSTND.BOOT (4 blocks) — Contains bootstrap code for RTSTND.APPLE.
* RTBSTRP.BOOT (4 blocks) — Contains bootstrap code for RTSTRP.APPLE.
* II40.MISCINFO (1 block) — Miscinfo file optomized for a 40-column Apple II or Apple II Plus. Identical to that supplied with the Development System.
* II80.MISCINFO (1 block) — Miscinfo file optomized for an 80-column Apple II or Apple II Plus . Identical to that supplied with the Development System.
* IIE40.MISCINFO (1 block) — Miscinfo file optomized for a 40-column Apple //e. Identical to that supplied with the Development System.
* SYSTEM.MISCINFO (1 block) — Miscinfo file optomized for an 80-column Apple //e. Identical to that supplied with the Development System.
* SYSTEM.CHARSET (2 blocks) — Identical to that supplied with the Development System, it is included here only for redundancy's sake. SYSTEM.CHARSET is needed on the Vendor Product Diskette only if TURTLEGRAPHICS is used.

The following files are contained on "RT64:", the Apple II Pascal 1.2 64K Runtime System diskette:

* SYSTEM.APPLE (32 blocks) — 64K Runtime "standard" P-machine.
* SYSTEM.PASCAL (29 blocks) — 64K Runtime operating system.

```
* SYSTEM.LIBRARY  ─┐
* SYSTEM.ATTACH    |
* RTSETMODE.CODE   |
* II40.MISCINFO    |——> same files as 48K Runtime System
* II80.MISCINFO    |
* IIE40.MISCINFO   |
* SYSTEM.MISCINFO  |
* SYSTEM.CHARSET  ─┘
```

The following files are contained on "RT128:", the Apple II Pascal 1.2 128K Runtime System diskette:

* SYSTEM.APPLE (32 blocks) — 128K Runtime "standard" P-machine.
* SYSTEM.PASCAL (29 blocks) — 128K Runtime operating system.

```
* SYSTEM.LIBRARY  ─┐
* SYSTEM.ATTACH    |
* RTSETMODE.CODE   |——> same files as 48K Runtime System
* SYSTEM.MISCINFO  |
* SYSTEM.CHARSET  ─┘
```

Of these files, the final Vendor Product Diskette should contain only the Runtime P-machine (RTSTND.APPLE, RTSTRP.APPLE, or SYSTEM.APPLE), SYSTEM.PASCAL, SYSTEM.LIBRARY, the appropriate miscinfo file renamed to SYSTEM.MISCINFO, and, optionally, SYSTEM.CHARSET. Information on the different miscinfo files is contained in the Apple II Pascal 1.2 Update Manual. SYSTEM.ATTACH, with its attendant data files as described in the Apple II Pascal Attach Tools manual, should be included on the Vendor Product Diskette if and only if special device drivers, written in machine-code, must be bound into the system for use by the Applications Package. All other files on the Runtime System diskettes are used in creating and configuring the Vendor Product Diskette.

IV. OPERATION

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

Note that the main application program must be named SYSTEM.STARTUP, so that the Runtime System can find it at bootstrap-load time.

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

A. The Bootstrapping Process

In a machine equipped with an auto-start ROM, the bootstrap loading process occurs automatically, as soon as the Apple's main power switch is turned "ON." As a result, the end-user is greeted by the applications package. In a machine that lacks an auto-start ROM, the end-user first encounters the Apple MONITOR, or BASIC, and must initiate the bootstrapping process by issuing a 6-CTRL-P command (in the case of the MONITOR) or a PR#6 command (for BASIC).

The bootstrap loader checks for the P-machine file and loads it into RAM. The P-machine, in turn, brings in and initializes the Runtime operating system. (In the case of a two-stage boot, the message "Insert boot diskette with SYSTEM.PASCAL on it, then press RETURN" appears after the P-machine has been loaded. The end-user should then insert the second-stage boot diskette and press RETURN which results in the Runtime 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 Diskette. Remember that SYSTEM.ATTACH must not be present on the Vendor Product Diskette unless special, low-level I/O drivers must be bound into the system. As explained more fully in the Apple II Pascal Attach Tools manual, SYSTEM.ATTACH uses two special data files, and will fail if these files are not present on the bootstrap diskette. A vendor who puts SYSTEM.ATTACH on his Vendor Product Diskette without also providing the data files required by that program insures consistent failure of the system bootstrap process. The vendor may include the SYSTEM.ATTACH software on the Vendor Product Diskette, while defeating the automatic execution of that utility at bootstrap load time, by changing its name in the diskette directory.

The bootstrap process culminates when the main applications program, SYSTEM.STARTUP, is loaded and executed. Any failure during the bootstrap process is fatal. Whenever possible, a failure will leave displayed the message

SYSTEM FAILURE NUMBER nn. PLEASE REFER TO PRODUCT MANUAL.

Here, "nn" refers to the actual number reported when the failure occurs. This number will correspond 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 ennumerate and explain for the end-user the situations in which bootstrap failures may occur, as well as suggest remedies for these failures.

B. 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 overlaid, or 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 applications 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 applications system terminates without chaining to another one, the Runtime system re-boots into SYSTEM.STARTUP.

VI. SPECIFICATIONS

A. Available Configurations

The memory requirements of different applications impose the need for different Runtime Systems. The applications 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 applications package. Apple currently supports the following Runtime Systems:

* 48K Runtime System (standard and stripped versions)
* 64K Runtime System (standard version only)
* 128K Runtime System (standard version only)

The difference between the standard and stripped versions of the 48K

Runtime 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 Runtime 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 3668 bytes.

FREE MEMORY IN APPLE II PASCAL 1.2 RUNTIME SYSTEMS

	NO SWAPPING	SWAPPING ON LEVEL 1
48K STANDARD	23372 bytes	27040 bytes
48K STRIPPED	25676 bytes	29344 bytes
64K	40322 bytes	43990 bytes
128K (CODE)	41227 bytes	44879 bytes
128K (DATA)	44502 bytes	44526 bytes

NOTE - the amount of free memory available with the 64K Runtime System is reduced by 1024 bytes if it is operating in 40-column mode.

There is another level of swapping (level 2) which provides an additional 822 bytes of usable memory, however, application writers should not depend on the extra memory being available in the future. Certain planned enhancements to the Pascal system will reduce the memory available to applications by approximately 1000 bytes. Swapping level 2 will help programs currently running at the limit of available memory to run under the enhanced system.

NOTE - using GET or PUT to disk will be slow if swapping level 2 is selected since these routines will have to be loaded 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 1.2 Update Manual for further information on swapping.

B. Use Environment

The hardware environment must include the following:

48K Runtime System	- An Apple II or II Plus with 48K of RAM (minimum), or an Apple //e
64K Runtime System	- An Apple II or II Plus with 48K of RAM and an Apple Language Card, or an Apple //e
128K Runtime System	- An Apple //e with an Extended 80-column Text Card
All Runtime Systems	- At least one disk drive, set up for 16-sector operation.
All Runtime Systems	- Video screen or external terminal (video screen preferred).

Note that the Runtime Systems support all Apple peripheral cards. Other cards may not operate properly, especially if they include firmware that depends upon specific internal characteristics of the P-machine interpreter 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, at each and every boostrap load. Note that the addition of special I/O drivers to the system will further restrict 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 the Apple II Pascal Attach Tools manual.

C. 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 Runtime 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 Runtime System diskettes should be used in the Runtime execution environment. (Furthermore, 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 Runtime 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. So, TURTLEGRAPHICS may be used under the 48K Stripped Runtime System if and only if the turtle is allowed to make only right-angle turns (as in the HILBERT demonstration program on APPLE3:, for example). Attempts to draw arbitrary curves, as demonstrated in the GRAFDEMO program on APPLE3:, will produce execution errors in the 48K Stripped Runtime environment.

3. Pascal's special function keys retain their meanings in the Runtime Systems. The following keys have special meaning:

   * Freeze (Stop) screen display - CTRL-S
   * Flush screen display - CTRL-F
   * Switch to alternate half of screen - CTRL-A
   * Toggle display to switch screen halves to follow cursor - CTRL-Z
   * Left square bracket - CTRL-K
   * Right square bracket - SHIFT-M
   * Break - CTRL-@
   * Upper/lower case activation toggles - CTRL-W, CTRL-E

   NOTE - Some of these special function keys are ignored by Pascal if it is running on a //e. See the Apple II Pascal 1.2 Update Manual for more information. It is possible to disable some of these special key functions. See the Apple II Pascal Attach Tools manual for complete details.

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

5. The Runtime System is optimized for operation with the Apple's built-in video output screen. There is no easy way for a turnkey package to reconfigure its host Runtime 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 clearscreen, random cursor addressing, or backspacing, is not likely to work properly on an external terminal. To avoid this problem, the Runtime 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.

D. Runtime System Configuration Utilities

1. RTSETMODE (provided with all Runtime Systems)

Flags which note the state of four system options are contained within a special part of the directory of any Runtime System bootstrap diskette. (These flags will not normally be present on diskettes 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 bootstrap time, the option-flags are retrieved and are used during a dynamic configuration process which occurs before the applications software is executed.

The RTSETMODE utility is used by the applications developer to set or reset the option-flags, according to the requirements of the applications 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 "Y" to any of the following questions ARMS the indicated option (setting the corresponding flag), while answering "N" DISARMS the option (and resets the corresponding flag).

* ARM Filehandler Overlay Option? - Arming this option sets swapping to level 1. System primitives related to disk file opening and closing are overlaid as needed by the application software, thus freeing 3668 bytes of RAM for use by the application.

* ARM Single-Drive System Option? - With this option armed, once the initial bootstrap process is finished at the beginning of any turnkey software run, the system itself will not assume the availability of any disk drives other than the bootstrap device. Specifically, "volume searches" will be limited to the single 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 system CONSOLE: device will always be the Apple's built-in video screen, whether or not an external terminal interface or 80-column card is available in slot 3.

* ARM Get/Put and Filehandler Overlay Option? - Arming this option sets swapping to level 2. System primitives related to disk file opening and closing, as well as GET and PUT to disk are overlaid as needed. (See section A 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 Diskette. If so, an attempt is made to update the diskette'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 Runtime bootstrap diskette will retain its option-flags. Transferring the Runtime System and applications software from diskette to diskette on a file-by-file basis will not also transfer the option-flags between the diskettes. For this reason, it is recommended that RTSETMODE be applied to the product master of any Runtime-based package immediately prior to releasing that master to production, in order to insure the correct status of the option-flags.

If a two-stage boot will be used for a runtime application, RTSETMODE must be run on both boot diskettes since some of the flags are checked by the P-machine while others are checked by the operating system.

2. RTBOOTLOAD (48K Runtime System only)

This program is used to transfer to the Vendor Product Diskette the proper bootstrap code for the chosen 48K Runtime 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 bootstrap code (either RTBSTND.BOOT or RTBSTRP.BOOT). The file name 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 file name. Once it has fetched the bootstrap code, RTBOOTLOAD asks for the volume name of the Vendor Product Diskette, then waits for the user to press the SPACE-BAR (thus providing the user with an opportunity to mount the selected volume, if necessary) before attempting to transfer the bootstrap information. The success or failure of the transfer is reported before RTBOOTLOAD terminates. This program is only supplied on the 48K Runtime System diskette and should never be used to transfer bootstrap information to a diskette which contains the 64K or 128K Runtime Systems, as doing so will prevent the systems from booting correctly.

E. Error Handling

If an error in execution or I/O occurs during program operation, the Runtime System attempts to let the application package itself acknowledge, and if possible, recover from the error condition. Just as he may in 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 Runtime 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 applications 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 Runtime 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 Runtime execution:

00 No error	100 Unknown Runtime error
01 Bad block, parity error	101 Value range error
02 Bad I/O 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	111 Unimplemented instruction
12 File already open	112 Floating point error
13 File not open	113 String overflow
14 Bad input format	114 Programmed HALT
16 Disk is write-protected	115 Programmed breakpoint
17 Illegal block number	116 Codespace overflow
18 Illegal buffer address	
19 Must read a multiple of 512 bytes	
20 Unknown ProFile error	
64 Device error (bad disk format)	

* = fatal error

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 re-boot 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 applications 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 Runtime 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 Runtime system detects the end of the standard input file (EOF), and re-boots (right back into "Calculator").

V. DIFFERENCES BETWEEN THE PASCAL DEVELOPMENT SYSTEM AND THE RUNTIME SYSTEMS

Although the Runtime Systems will run most Pascal code files exactly as does the Pascal Development System, the applications system 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 Runtime System due to the difference in available user memory. Similarly, programs developed with the 128K Development System may fail to execute under the 64K Runtime System for the same reason. While large systems can be made to fit within the confines of a particular Runtime 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 and/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 applications package runs the risk of massive failure unless calls to program overlays and chaining are preceded by checks that the expected diskette 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 Runtime Systems:

* System HOMECURSOR, CLEARSCREEN, and CLEARLINE functions
* System prompt function
* Compiler, Assembler, Linker, Editor, Filer
* IDSEARCH and TREESEARCH procedures (which exist in the Development System only to benefit the Compiler).

Programs that make use of information stored in specific memory locations within the 1.2 Development System P-machine, or that make assumptions about static or dynamic memory allocation at the operating system level (e.g., for the purpose of accessing system data structures) are likely to function incorrectly when executed in the Runtime environment. This is due to the code reorganization, compaction, and optimization that was necessary to produce the Runtime Systems.

VII. CREATION OF VENDOR PRODUCT DISKETTE

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

1 - Format a diskette 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 Runtime System diskette to the Vendor Product diskette.

3 - Transfer the code file(s) for the application to the Vendor Product diskette. 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 diskette.

5 - Run RTBOOTLOAD to load the appropriate bootstrap code from RT48: onto the Vendor Product diskette. (48K RUNTIME SYSTEMS ONLY)

6 - Run RTSETMODE if you wish to ARM the "Filehandler Overlay" option, the "Single-Drive System" option, the "Ignore External Terminal" option and/or the "Get/Put and Filehandler Overlay" option.

Vendor Product Diskettes, or other diskettes which contain 48K Runtime 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 "#5:=" to "#5:$"), will only copy files from one disk to another. They will not copy the crucial 48K Runtime boostrap code between disks. Only "whole volume" transfers (such as "#4:" to "#5:", or "SOUP:" to "NUTS:") will result in complete copies, containing the proper bootstrap information.

Vendor Product Diskettes, or other diskettes which contain 64K or 128K Runtime System software can be copied using either whole volume or individual file transfers since they do not contain special bootstrap information.

VIII. APPLE FORTRAN AND THE RUNTIME SYSTEMS

Apple FORTRAN programs will execute correctly under the Apple II Pascal 1.2 Runtime Systems (48K and 64K only), so 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 Runtime 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 re-boot in the Runtime 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 Runtime Systems require for robust execution of turnkey software.

APPLE COMPUTER, INC.
20525 Mariani Avenue
Cupertino, CA 95014

PASCAL TECHNICAL NOTE #11

Apple Pascal 1.1
BIOS Reconfiguration Using ATTACH

(02 April 1981)

For further information contact:
PCS Developer Technical Support
M/S 22-W, Phone (408) 996-1010

Disclaimer of All Warranties and Liabilities

Apple Computer, Inc. makes no warranties, either express or implied, with respect to this documentation or with respect to the software described in this documentation, its quality, performance, merchantability, or fitness for any particular purpose. Apple Computer, Inc. software is sold or licensed "as is". The entire risk as to its quality and performance is with the vendor. Should the programs prove defective folowing their purchase, the vendor (and not Apple Computer, Inc., its distributor, or its retailer) assumes the entire cost of all necessary servicing, repair, or correction and any incidental or consequential damages. In no event will Apple Computer, Inc. be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the software, even if Apple Computer, Inc. has been advised of the possibility of such damages. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation may not apply to you.

This documentation is copyrighted. All rights are reserved. This document may not, in whole or part, be copied, photocopied, reproduced, translated or reduced to any electronic medium or machine readable form without prior consent, in writing, from Apple Computer, Inc.

Copyright 1981 by Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014
(408) 996-1010

Notice

Apple Computer, Inc. reserves the right to make improvements in the product described in this document at any time and without notice.

ATTACH-BIOS document for Apple II Pascal 1.1

By Barry Haynes

Jan 12, 1980

This document is intended for Apple II Pascal internal applications writers, Vendors and Users who need to attach their own drivers to the system or who need more detailed information about the 1.1 BIOS. It is divided into two sections, one explaining how to use the ATTACH utility available through technical support and the other giving general information about the BIOS. It is a good idea to read this whole document before assuming something is missing or hasn't been completely explained. This document is intended for more advanced users who already know a fair amount about I/O devices and how to write device drivers. It is not intended to be a simple step by step description of how to write your first device driver, nor does it claim to be a complete description of all there is to know about the Pascal BIOS.

The Apple Pascal UCSD system has various levels of I/O that are each responsible for different types of actions. It was divided at UCSD into these levels to make it easy to bring up the system on various processors and also various configurations of the same processor and yet have things look the same to the Pascal level regardless of what was below that level. The levels are:

LEVEL	TYPES OF IO ACTIONS
Pascal	READ & WRITE BLOCKREAD & BLOCKWRITE UNITREAD & UNITWRITE UNITCLEAR UNITSTATUS
RSP (Runtime Support Package)	This is part of the interpreter and is the middle man between the above types of I/O and the below types of I/O. All the above types are translated by the compiler and operating system into UNITREAD, UNITWRITE, UNITCLEAR and UNITSTATUS if they are not already in that form in the Pascal program. The RSP checks the legality of the parameters passed and reformats these calls into calls to the BIOS routines below. The RSP also expands DLE (blank suppression) characters, adds line feeds to carriage returns, checks for end of file (CTRL C from CONSOLE:), monitors UNITRW control word commands, makes

	calls to attached devices if present, echoes to the CONSOLE:.
BIOS (Basic I/O Subsystem)	This is the lowest level device driver routines. This is the level at which you can attach new drivers to replace or work with the regular system drivers and also attach drivers for devices that will be completely defined by you.

I.RECONFIGURING THE BIOS TO ADD YOUR OWN DRIVERS USING THE ATTACH UTILITY.

INTRODUCTION

With the Apple Pascal 1.1 System (both regular and runtime 1.1), there is an automatic method for you to configuer your own drivers into the system. This method requires you to write the drivers following certain rules and to use the programs ATTACHUD.CODE and SYSTEM.ATTACH provided through Apple Technical Support. At boot time, the initialization part of SYSTEM.PASCAL looks for the program SYSTEM.ATTACH on the boot drive. If it finds SYSTEM.ATTACH, it Xecutes it before Xecuting SYSTEM.STARTUP. SYSTEM.ATTACH will use the files ATTACH.DATA and ATTACH.DRIVERS which must also be on the boot disk. ATTACH.DATA is a file the develcper will make using the program ATTACHUD. It tells SYSTEM.ATTACH the needed information about the drivers it will be attaching. ATTACH.DRIVERS is a file containing all the drivers to be attached and is constructed by the developer using the standard LIBRARY program. The drivers are put on the Pascal Heap below the point that a regular program can access it. They do take away Stack-Heap (= to the size of the drivers attached) space from that available to Pascal code files but this should not be a problem unless the drivers are very large or the code files very hungry in their use of memory. Since these drivers are configured into the system after the operating system starts to run, this method will not work for configuring drivers for devices that the system must cold boot from. Some of supporting code in the RSP, boot and Bios may make the task of bringing up boot drivers easier though. The advantages to this kind of setup are:

1. Software Vendors can use the ATTACHUD program to put their own drivers into the system at boot time. This will be invisible to the user.

2. There can be no problems losing drivers due to improper heap management since the drivers are put on the heap by the operating system and before any user program can allocate heap space.

3. This method does not freeze parts of the system to special memory locations since it enforces the clean methodology of using relocatable drivers.

USING ATTACHUD

ATTACHUD.CODE will ask you questions about the drivers you want to attach to the system. It makes a file called ATTACH.DATA which tells SYSTEM.ATTACH which drivers to attach to the system, what unit numbers to attach them to and other information. The options covered by ATTACHUD are:

1. A driver can be attached to one of the system devices, then all I/O to this device (PRINTER: for example) will go to this new driver. In the case of a new driver for a disk device the user will have to specify which of the 6 standard disk units will go to this new driver. This will allow replacement of standard drivers with custom ones without having to restrict the I/O interface to UNITREAD and UNITWRITE as is the case with option 2.

2. A driver can be attached to one of 16 userdevices. I/O to these will be done with UNITREAD and UNITWRITE to device numbers 128-143.

3. A method will be included to allow the attached driver to start on an N byte boundry. The driver writer will be responsible for aligning his code from that point.

4. More than one unit can be attached to the same driver. This way only one copy of the driver resides in memory and I/O to all the attached units goes to this one driver. It is up to the driver to decide which unit's I/O it is doing. How this is done is explained below.

5. The initialize routine for any attached driver can be called by SYSTEM.ATTACH after it has attached the driver and before any programs can be Xecuted.

6. In case any of your programs use the Hires pages, you can specify in ATTACHUD that drivers must not be put on the heap over these areas. Your drivers would have to be quite large before they could possibly overlap the Hires pages.

Follow through this example of a session with ATTACHUD where the options available are completely described. First Xecute ATTACHUD:

You will be given the prompt:

```
Apple Pascal Attachud [1.1]

Enter name of attach data file:
```

This is asking for what you want the output file from this session with ATTACHUD to be called. You could call it ATTACH.DATA or some other name and then rename it to ATTACH.DATA when you put it on the boot disk with SYSTEM.ATTACH.

If you ever get a message of the form:

```
ERROR => some error
Try again (RETURN to exit program):
```

then just retype what was requested on the previous prompt after deciding what mistake you made while typing it the first time.

The next prompt is:

```
These next questions will determine if
attached drivers can reside in the hires
pages.  It will be assumed they can for the
page in question if you answer no to the
prompt for that page.
Will you ever use the (2000.3FFF hex)
hires page?
```

Followed by:

```
Will you ever use the (4000.5FFF hex)
hires page?
```

You should answer yes to the question for a particular Hires page if you will ever be running a program that uses that Hires page while the drivers are Attached. You don't want the possibility of your driver residing in the Hires page if that page will be clobbered by one of your programs. After answering the Hires questions you will be asked the following questions once for each driver you will be attaching:

```
What is the name of this driver?  This
must be the .PROC name in its assembly
source (RETURN to exit program):
```

This must be the name of one of the drivers in the ATTACH.DRIVERS that will be used with this ATTACH.DATA. The length of this name must not be more than 8 characters. After entering the name you will be asked:

```
Which unit numbers should refer to this
device driver?

Unit number (RETURN to abort program):
```

You must enter a unit number in the range 1,2,4..12,128..143 or will be given an error message. You cannot attach a character unit (CONSOLE:, PRINTER: or REMOTE:) to the same driver as a block structured unit and if you try you will be given the message:

```
You can''t attach a character unit and
a block unit to the same driver.  I
will remove the last unit# you entered.
Type RETURN to continue:
```

If you don't get the above error, you will be asked:

```
Do you want this unit to be
initialized at boot time?
```

A yes response will put the unit number just entered on a list of units that SYSTEM.ATTACH will call UNITCLEAR on after attaching all

the drivers. This gives you a way to have the system make an initialize call on your attached unit at boot time. A no response will mean that no boot time init call will be made on this unit to the driver you just attached.

You will be eventually asked:

```
Do you want another unit number to refer
to this device driver?:
```

A yes response will get you to the Unit number prompt again and a no response will get you to the prompt:

```
Do you want this driver to start on a
certain byte boundary?
```

A yes here will give you more prompts:

```
The boundry can be between 0 and 256.
  0=>Driver can start anywhere.(default)
  8=>Driver starts on 8 byte boundary.
  N=>Driver starts on N byte boundary.
256=>Driver starts on 256 byte PAGE boundary.
Enter boundary (RETURN to exit program):
```

And the last line of the prompt will repeat until you enter a boundary in the correct range. The boundary refers to the memory location where the first byte of the driver is loaded. If your driver needs to be aligned on some N byte boundary you can assure it will be using this mechanism. if you know how the driver's origin is aligned, You can align internal parts of your driver however you want. Finally you will get to the prompt:

```
Do you want to attach another driver?
```

And if you answer Yes to this you will return to the 'What is the name of this driver' prompt and answering No will end the program, saving the data file you have made.

THE DRIVER

Drivers must be written in assembly using the Pascal Assembler. They must not use the .ABSOLUTE option, so the drivers can be relocated as they are brought in by the system. Each driver must be assembled separately with no external references. When all drivers are assembled, use the LIBRARY program (in the same way you would use it to put units into a library) to put all the drivers in one file. Name this file SYSTEM.DRIVERS. See further explanation of making SYSTEM.DRIVERS below.

Considerations for all drivers:

1. Study the examples below as certain information is only documented there.

2. Refer to the Apple II Pascal memory map below and you will see

that parts of the interpreter and BIOS reside in the same address range and are bank-switched. The system automatically folds in the BIOS for drivers added using ATTACH. Most of these drivers will have to make calls to CONCK if they want type ahead to continue to work properly. CONCK is the BIOS routine that monitors the keyboard. See the example drivers below to be sure you are doing this correctly. You cannot call CONCK through the CONCK vector at BF0A (see BIOS part of this document) because this call would go through the same mechanism used to get to your driver and the return address to Pascal would be lost.

3. All attached drivers must be written with one common entry point for read, write, init and status. The driver will use the Xreg contents to decide which type of I/O call this is and jump to the appropriate place within it's code. The Xreg is decoded as follows:

   0 -->read (no bits set)
   1 -->write (bit 0 set)
   2 -->init (bit 1 set) § The Pascal statement
   UNITCLEAR(UNITNUMBER); makes an init call for
   unit UNITNUMBER †
   4 -->status (bit 2 set)

4. The drivers must also pop a return address off the stack, save it and later push it to do a RTS when the driver is finished. All other parameters must be removed from the stack by the driver. For all calls, the return address will be the top word on the stack.

5. SYSTEM.ATTACH will make a copy of the normal system jump vector (the vector after the fold) and put this on the heap. There will be a pointer to this vector at 0E2. Your drivers can use this vector to get to the normal system drivers for device numbers 1..12. See example below.

6. All drivers must pass back a completion code in the X register corresponding to the table on page 280 of the 1.1 "Apple II Apple Pascal Operating System Reference Manual".

7. In references below to parameters passed on the stack, all parameters are one word parameters so they require two bytes to be popped from the stack by the driver.

8. Control word format for Unitread & Unitwrite

bits	15..13	12..6	5	4	3	2	1..0
	user defined functions	reserved for future expansion	type B chars	type A chars	nocrlf	nospec	reserved for future expansion

type B =0 ==>System will check for CTRL S & F from CONSOLE: during the time of this Unitio call.
=1 ==>System will not check for CTRL S & F during this Unitio.
type A =0 ==>If using Apple Keyboard, system will check for CTRL A,Z,K,W & E from CONSOLE: during the period of this Unitio.

```
             =1 ==>System will not check for the chars during
                   this Unitio.
      nocrlf =0 ==>line feeds are added to carriage returns by the
                   Interpreter.
             =1 ==>no line feeds are added ...
      nospec =0 ==>DLE's (blank suppression code) are expanded on
                   output and the EOF character is detected on input
             =1 ==>nothing special is done to DLE's on output and
                   EOF on input.

      default setting for all control word bits = 0.
```

9. Control word format for UNITSTATUS

```
   bits  15..13    12..2      1          0
         user      reserved   for        direction
         defined   for future purpose

         direction =0 ==>Status of output channel is requested
                   =1 ==>Status of input   ...
         purpose   =0 ==>Call is for unit status
                   =1 ==>Call is for unit control
```

10. These are the new vectors and routines added to the BIOS to make attach work. The RSP, bootstrap, and readseg were also modified to allow for attaches.

```
UDJMPVEC   ;Jump vector for user devices, offset=0 => unattached device.
           ;The correct addresses are initialized by SYSTEM.ATTACH
           ;See locations section of BIOS part below for pointers to
           ;this vector.
           JMP     0              ;Unit 128
           JMP     0              ;Unit 129
           .
           .
           .
           JMP     0              ,Unit 143

DISKNUM    ;If high byte=FF then
           ;  device is not a disk drive
           ;else
           ;  if high byte=0 then
           ;    device is a regular disk drive and low byte=drive #
           ;  else
           ;    driver for this disk drive has been attached by SYSTEM.ATTACH
           ;    and the driver address is stored in this word.
           ;    (Driver address has to be the address-1 for RTS in PSUBDR
           ;     to work correctly, remember this for ATTACH.  PSUBDR is
           ;     listed below.)
           ;See locations section of BIOS part below for pointers to
           ;this vector.
           .WORD   0FFFF          ;Unit #1
           .WORD   0FFFF          ;Unit #2  (ATTACH would modify the words
           .WORD   0FFFF          ;Unit #3   for units 4,5,9..12 if a
           .WORD   0              ;Unit #4   different disk driver were
           .WORD   1              ;Unit #5   attached to any of them)
           .WORD   0FFFF          ;Unit #6
```

```
        .WORD   0FFFF           ;Unit #7
        .WORD   0FFFF           ;Unit #8
        .WORD   4               ;Unit #9
        .WORD   5               ;Unit #10
        .WORD   2               ;Unit #11
        .WORD   3               ;Unit #12

UDRWIS  ;Routine to get to an attached driver through UDJMPVEC
        ;Assume unit# in Areg & operation to be performed in Xreg.
        ;See the jump vector in the BIOS sections to see how you
        ;get to this routine.
        STA     TT1
        AND     #7F             ;Clear top bit of unit#
        STA     TT2             ;Make address in UDJMPVEC table
        ASL     A               ;Address=Areg*3 + base of table
        CLC
        ADC     TT2             ;Now we have (Areg*3).
        ADC     #JVECTRS        ;Add in low byte of base of table having
        STA     TT2             ;no carry problem with only 16 UD's.
        LDA     #0
        ADC     JVECTRS+1       ;JVECTRS is a word pointing to the base
                                ;of UDJMPVEC.
        STA     TT2+1
        LDA     TT1
        JMP     @TT2


PSUBDR  ;Routine to get to an attached driver through DISKNUM
        ;We assume on entry, Areg=unit#, Yreg=DISKNUM
        ;offset & Xreg=the command to be performed by the substituted
        ;disk driver.
        ;See the jump vector in the BIOS sections to see how you
        ;get to this routine.
        STA     TT1           ,Save unit#.
        LDA     DISKNUM-1,Y ;Store MSB of driver address.
        PHA
        LDA     DISKNUM-2,Y ;Store LSB of driver address.
        PHA
        LDA     TT1           ;Restore unit# to Areg.
        RTS                   ;Jump to substituted driver.  This assumes
                              ;the driver address in DISKNUM =
                              ;(ADDRESS OF DRIVER)-1 for the RTS to work
```

Special considerations when attaching drivers for the system devices, unitnumbers 1..12.

A. Character Oriented Devices (Pass the character to be read-written in the A-register and make Bios calls one character at a time from RSP level. On entry, the unit number will be in the Y register in case you wanted to attach all character oriented devices to the same driver). If you attach REMOTE: & or PRINTER: to the same driver as CONSOLE:, all will have their jump vectors pointing to the start of the driver+3 bytes. See further discussion on this below.

Units 1 & 2 (CONSOLE: and SYSTERM:)

1. These must both go to the same driver.

2. The system CONCK routine will be patched to jump to the start of the driver. The CONCK routine gets characters entered at the keyboard and fills the type ahead buffer. See the example CONSOLE: driver below.
3. Because of item 2, the entry point for normal calls (not CONCK calls) to the attached driver will be 3 bytes beyond the start of the driver.
4. The interpreter takes care of expanding blank suppression codes (DLE's), echo to the screen, EOF (the end of file character), and adding line feeds to every carriage return. Your driver doesn't need to do this.
5. CONSOLE: read and write have only the return address on the stack. The stack for CONSOLE: init looks like:

```
        POINTER TO BREAK VECTOR (This should be stored at
                                 location BF16..BF17 by CONSOLE:
                                 init.)
        POINTER TO SYSCOM       (This should be stored at
                                 location F8..F9 by CONSOLE:
                                 init.)
                                (Also at init time, the Flush
                                 and Start/Stop conditions
                                 should be set to normal and
                                 the type-ahead queue should
                                 be emptied.)
        RETURN ADDRESS         <--TOS (top of stack)
```

The stack for CONSOLE: status looks like:

```
        POINTER TO STATUS RECORD
        CONTROL WORD
        RETURN ADDRESS         <--TOS
```

6. A status request should return, in the first word of the status record, the number of characters currently queued in the direction asked for. This is the number of characters in the type-ahead buffer. If no type-ahead is being used then output status should always return a 0 and input status a 1 if a char is waiting to be read, otherwise a 0.
7. Since we are using 7 bit ASCII codes, the CONSOLE: read routine should zero the high order bit of all characters it reads from the keyboard and passes back to Pascal (to the RSP). The CONSOLE: write routine should transfer all 8 bits as received from the RSP since many devices use 8 bit control codes.
8. The RSP will send both upper and lower case chars to the CONSOLE: write routine. The write routine should map the lower to upper if the device cannot handle lower case.
9. CONSOLE: Output Requirements:
   A. CR (0D hex) A carriage return should move the cursor to the beginning of the current line.
   B. LF (0A hex) A line feed should move the cursor to the next line but not change the column position. If the cursor is on the last line on the screen when a line feed is sent, the rest of the screen should scroll up one line and the bottom line be cleared.
   C. BELL (07 hex) A sound should be made if possible when the CONSOLE: gets 07. If making a sound is not possible then ignore the 07.
   D. SP (20 hex) Place a space at the current cursor position overwriting whatever is there. Move the cursor to the next column. If the cursor is on the last column of a line, it is best if the cursor stays where it is after the space fills that

position. If the cursor is on the last column of the last line on the screen, it is also best if the cursor remains in that position and the screen does not scroll. These are the prefered actions of the cursor at end of line & end of screen; in the strict sense, the actions of the cursor in these circumstances are undefined.

E. NUL (00 hex) When a Null is sent to the CONSOLE: from the RSP, the CONSOLE: should delay for the ammount of time required to write one character but the state of the screen should not change.

F. All printable characters should be written to the screen and the cursor should move in the same way it does for SP.

G. See the discussion on pages 199-215 in the 1.1 Operating System Reference Manual for further requirements and information.

10. CONSOLE: Input Requirements:

A. The RSP takes care of echoing characters to the screen typed from the CONSOLE: keyboard.
(below items optional The Start/Stop, Flush & Break chars are redefinable; see 9G above for more info.)

B. The Start/Stop character is detected by CONCK and is used to stop all processing until the character is received a second time. When the character is received (see 9G above for more info) one should loop in CONCK continuing to process other characters until:
   1. the S/S char is received again
   2. the Break char is received
In case 1, the suspended processing should continue as it was before the first S/S was typed. Action needed for the Break char is described below. The S/S char is never returned to the RSP and CONSOLE: type-ahead, if implemented, should continue during the suspended state. Offset from SYSCOM to this char is 85 decimal. (This and the next 2 chars are redefinable by the Setup program and SYSCOM is the system area that keeps track of this info. The pointer to the start of SYSCOM is passed to the CONSOLE: init routine and is stored at F8..F9 hex.)

C. The Flush character will stop all output and echoing to the CONSOLE: until it's second occurEnce (see 9G above). CONCK detects this and must set a flag to tell the CONSOLE: output routine to ignore characters while the flag is set. If the CONSOLE: is re-initialized or a Break_char is received, the flush state should be turned off. Flush is never returned to the RSP. Flush only stops CONSOLE: output, other processing continues. Offset from SYSCOM to this char is 83 decimal.

D. The Break char should cause CONCK to jump to the location stored at BF16. This location is also passed to the CONSOLE: init routine which stores it at BF16. The break char is never returned to the RSP and it should remove the system from Stop or Flush mode if it is in either mode. Offset from SYSCOM to this char is 84 decimal.

E. Type-ahead should be implemented in CONCK by storing characters typed at the keyboard in a queue until they are requested by a CONSOLE: read from Pascal. When the queue fills, further characters should be ignored and a bell sounded when they are typed. The length of the queue should be at least 20 characters.

11. For more information on CONSOLE: requirements, see pages 199-

216 of the 1.1 Operating System Reference Manual.

Unit 6 (the PRINTER:)

1. The interpreter takes care of expanding blank suppression codes (DLE's), EOF (the end of file character), and adding line feeds to every carriage return.
2. PRINTER: read,write and init have only the return address on the stack. PRINTER: status has the same items on the stack as CONSOLE: status. PRINTER: init should cause the PRINTER: to do a carriage return and a line feed and throw away any characters buffered to be printed. No form feed should be done.
3. For status, return in the first word of the status record the number of bytes buffered in the direction asked for; if this cannot be determined by your PRINTER:, return a 0.
4. The PRINTER: write routine must buffer a line and send it all at once if your PRINTER: can only receive data that way.
5. Line Delimiter characters:
   A. CR (hex 0D) A carriage return should cause the PRINTER: to print the current line and return the carriage to the first column. An automatic line feed should not be done by the PRINTER: driver when it reads a CR.
   B. LF (hex 0A) The RSP will send line feeds to the PRINTER: driver after each carriage return. This should cause the PRINTER: to advance to the next line. If the PRINTER: must also do a carriage return when it is given a line feed, then this is O.K.
   C. FF (hex 0C) This should cause the PRINTER: to move the paper to top of form and do a carriage return. If top of form is not possible on your PRINTER:, do a carriage return followed by a line feed.
6. It is assumed that input cannot be received from the PRINTER:. See the BIOS section for a discussion of how to get input from the PRINTER:. Normally, trying to get input from the PRINTER: should return completion error code #3.

Units 7 (REMOTE: in) & 8 (REMOTE: out)

1. These must both go to the same driver.
2. The interpreter takes care of expanding blank suppression codes (DLE's), EOF and adding line feeds to every carriage return.
3. Same stack setup as the PRINTER:.
4. Status should return in first word of status vector the number of bytes buffered for the direction specified in the control word, 0 if you have no way to check.
5. This unit is supposed to be an RS-232 serial line for many different applications so it is necessary that it transfer the data without modifying it in any way. The transfer rate default is 9600 baud.
6. It would be nice if the input to REMOTE: could be buffered in the same way input to the CONSOLE: is but this is not an absolute requirement.
7. REMOTE: init should set up the REMOTE: device so it is ready to read and write.

B. Block Structured Devices

Units 4 (the boot unit),5,9,10,11,12.

1. These units are assumed to be block structured devices, the drivers for these units must do their own Pascal Block to Track-Sector conversions.
   The UCSD system assumes the disk device is a 0-based consecutive array of 512 byte logical blocks. All UCSD Pascal disks must have this logical structure no matter what their actual physical structure or size are. The physical allocation schemes for information on different types of disks are arranged with sectors that are of various sizes that depend on the hardware of the particular disk device used. The driver must convert the Pascal block # to the appropriate track & sector # of where that block is stored on it's disk device. This could be a floppy or hard disk or some other type of device. It doesn't really matter, so long as your driver maps the Pascal Block to the cOrrect place and continues to do so for the length (byte count) required for the UnitIO operation.
   The Pascal system uses logical blocks 0 & 1 for it's bootstrap code. These logical blocks should not be used for anything else and should therefore only be available to Pascal through direct UNITREAD & UNITWRITE operations and not accessable by the system through any other means. This document will not attempt to describe the boot sequence & does not attempt to give you enough information to attach another driver or device to unit #4: so you can cold boot from that unit.
   When a UNITWRITE is done to disk where the byte count MOD 512 is not equal to 0 (this means the last block included in the write would be partially written to according to the byte count), it is undefined whether garbage is written into the remaining part of this last block. So you may write a whole block anyhow if that is more efficient and the Pascal system will not suffer any bad consequences.
   When a UNITREAD is done from a disk you are not allowed to overwrite into the unused part of the last block (if there is an unused part due to byte count MOD 512 <> 0). You must only send the number of bytes asked for because you could clobber memory having some other valid use if you wrote extra bytes. You will have to buffer the last sector inside your disk read routine then transfer exactly the number of bytes from this last sector needed to add up to the total bytes requested.
2. The unit number will always be in the A register.
3. The stack setup for read and write is:

   ```
   CONTROL WORD      (The MODE parameter mentioned in the
                          1.1 Language Ref Manual on page 41)
   DRIVE NUMBER
   BUFFER ADDRESS
   BYTE COUNT
   BLOCK NUMBER
   RETURN ADDRESS  <--TOS
   ```

   For init there is only the return address on the stack and for status the setup is the same as for the CONSOLE:.
4. Status requests should return the following in the status record:
   - word1:Number of bytes buffered in the direction asked for in the control word. Return 0 if you have no way of checking.
   - word2:Number of bytes per sector.

word3:Number of sectors per track.
word4:Number of tracks per disk.

C. Other
Unit 3
1. This unit has no meaning for the Apple II system except that UNITCLEAR on this unit sets text mode.

Considerations when attaching drivers for user defined devices numbers 128-143.

These unit numbers are provided for you to do whatever you want with them. you can define what they do except for the following protocols.

1. Follow the considerations for all drivers listed above.
2. The unit number will always be in the A register.
3. The stack setup for read and write is:

```
CONTROL WORD
DRIVE NUMBER
BUFFER ADDRESS
BYTE COUNT
BLOCK NUMBER
RETURN ADDRESS  <--TOS
```

For init there is only the return address on the stack and for status the setup is the same as for the CONSOLE:.

This is a sample driver for a user defined device.

```
          ;Locations 0..35 hex may be used as pure temps. One should
          ;never assume these locations won't be clobbered if you leave
          ;the environment of the driver itself. ("leaving" includes
          ;calls to CONCK).

CONCKADR .EQU    02

          ;Only one .PROC may occur in a driver, each driver to be
          ;ATTACHED must be assembled separately using the Pasçal
          ;assembler and can have no external references.

          .PROC U128DR

          STA   TEMP1       ;Save Areg contents (unit#)
          PLA
          STA   RETURN
          PLA
          STA   RETURN+1
          TXA               ;Use the X reg to tell you what kind of
                            ;call this is.
          CMP   #2
          BEQ   INIT
```

```
        CMP    #4
        BEQ    STATUS
        CMP    #0
        BEQ    PMS
        CMP    #1
        BEQ    PMS

        ;Could have error code here
        JMP    RET

PMS     PLA              ;Get the parameters
        STA    BLKNUM
        PLA
        STA    BLKNUM+1
        PLA
        STA    BYTECNT
        PLA
        STA    BYTECNT+1
        PLA
        STA    BUFADR
        PLA
        STA    BUFADR+1
        PLA
        STA    UNITNUM   ;Also in TEMP1
        PLA
        STA    UNITNUM+1 ;Should always be 0
        PLA
        STA    CONTROL
        PLA
        STA    CONTROL+1
        TXA
        BNE    WRITE

READ    JSR    GOTOCK
        ;Your driver's code for a read
        (If more than one unit were attached to this driver, this
         code could jump to various places depending on the contents
         of the Areg stored in TEMP1)
        JMP    RET

WRITE   JSR    GOTOCK
        ;Your driver's code for a write
        JMP    RET

        ;If you wanted to call CONCK whenever your device did a read
        ;or write, you would use this routine:
CKR     .WORD CONCKRTN-1
GOTOCK  LDY    #55.      ;Offset to address of CONCK
        LDA    @0E2,Y
        STA    CONCKADR
        INY
        LDA    @0E2,Y
        STA    CONCKADR+1
        LDA    CKR+1     ;Set it up so you return to CONCKRTN after
        PHA              ;the CONCK call.
        LDA    CKR
        PHA
        JMP    @CONCKADR ,Jump to CONCK
```

```
CONCKRTN RTS            ;Return to caller.

INIT     ;Your driver's code for init
         JMP   RET

STATUS   PLA
         STA   CONTROL
         PLA
         STA   CONTROL+1
         PLA
         STA   BUFADR   ;Address of status record.
         PLA
         STA   BUFADR+1
         ;Your driver's code for status

RET      LDA   RETURN+1
         PHA
         LDA   RETURN
         PHA
         LDA   TEMP1
         RTS

RETURN   .WORD   0      ;Can't use 0 page for these since we leave
TEMP1    .WORD   0      ;our environment when going to CONCK.
CONTROL  .WORD   0
UNITNUM  .WORD   0
BUFADR   .WORD   0
BYTECNT  .WORD   0
BLKNUM   .WORD   0

         .END
```

This is a sample driver for a CONSOLE: driver replacement.

```
ROUTINE  .EQU  02
TEMP1    .EQU  04

         .PROC CKATCH

         JMP   CONCKHDL ;SYSTEM.ATTACH will patch the start of CONCK
                        ;to jump here when you attach a driver to the
                        ;CONSOLE:.

                        ;We are not popping the return address from
                        ;the stack cause we'll return from the system
                        ;routine we call from this driver.
         STA   TEMP1    ;All the read,write,init and stat calls will
                        ;jump here (the starting address of your
                        ;CONSOLE: driver+3).

         STY   TEMP1+1
         TXA
```

```
                        ;This example shows you how to have your
                        ;own code for the CONSOLE: as well as using
                        ;the system CONSOLE: routines.  If you want
                        ;to replace the system routines completely,
                        ;you need to pull the return address here.
         BEQ   READ
         CMP   #1
         BEQ   WRITE
         CMP   #2
         BEQ   INIT
         CMP   #4
         BEQ   STATUS

         ;Error code here

READ     ;Your driver's code for a read

         LDY   #1       ;offset to address of CONSOLE: read in
                        ;the copy of the jmp vector made by
                        ;SYSTEM.ATTACH.  See the jump vectors in the
                        ;BIOS section below to see how we get the
                        ;offsets.

         BNE   GET

         ;You would have a JMP RET here (see example for user defined
         ;device) if you were not using the system CONSOLE: routines
         ;as well.

WRITE    ;Your driver's code for a write
         LDY   #4
         BNE   GET

INIT     ;Your driver's code for init
         LDY   #7
         BNE   GET

STATUS   ;Your driver's code for status
         LDY   #43.

GET      LDA   @0E2,Y   ;At E2 is a pointer to the copy of the
                        ;jump vector made by SYSTEM.ATTACH before
                        ;it was modified to attach your drivers.

         STA   ROUTINE
         INY
         LDA   @0E2,Y
         STA   ROUTINE+1
         LDY   TEMP1+1  ;Restore registers
         LDA   TEMP1
         JMP   @ROUTINE ;Go to the original CONSOLE: driver for this
                        ;I/O command.  You will return from there; the
                        ;BIOS is already folded in due to the way your
                        ;driver was attached by SYSTEM.ATTACH.


CONCKHDL PHP            ;Duplicate the 1st three instructions of CONCK
         PHA            ;as they were patched by SYSTEM.ATTACH to jump
```

```
;TXA below      ;to the 1st instruction of this driver.

;Here you can put the code for your own part of CONCK (you
;may want to check some additional device like a keypad or
;something or you may want to replace the system CONCK
;routine alltogether.  If you do this, you must save the rest
;of the machine state and return it when you are finished.
;See example below.

TYA             ;Save Yreg contents for a second.
PHA

;This code gets us to the system CONCK routine.
CLC
LDY    #55.     ;Offset to the address of system CONCK in the
                ;copy of the original jmp vector.

LDA    @0E2,Y
ADC    #3       ;Add 3 so you enter right after the three
                ;instructions you duplicated at CONCKHDL.
STA    ROUTINE
INY
LDA    @0E2,Y
ADC    #0
STA    ROUTINE+1
PLA             ;Restore Yreg.
TAY
TXA             ;Last of CONCK instructions SYSTEM.ATTACH
                ;overwrote with the jmp to the start of this
                ;driver.

JMP    @ROUTINE ;Goto system CONCK and return from there.

.END
```

Here is another alternative for the CONCKHDL part of the above program.

```
CKRTN    .WORD CONCKRTN-1
CONCKHDL ; 1.If you don't care about type-ahead, this could be
         ;   simply the following code (assuming your CONSOLE:
         ;   read gets a character directly from your CONSOLE:
         ;   device whenever it is called) :

--------------------------------------------------------------
         PHP
         INC RANDL   ;RANDL is a permanent word at BF13 used in
                     ;the built in random function.
         BNE $1
         INC RANDH   ;RANDH
$1       PLP
         RTS
--------------------------------------------------------------

         ; 2.If you want type-ahead, this code should check to see
         ;if there is a character available and stuff it into a type-
         ;ahead buffer.
```

```
; 3.If you are using this with the regular CONCK (extra keypad
;to check or statistics for example), then you can do it this
;way.

-------------------------------------------------------------
         PHP            ;Save state of machine
         PHA
         TXA
         PHA
         TYA
         PHA

         ;Put your driver's part of CONCK here (gives your driver
         ;priority)

         LDA CKRTN+1 ;Set up things to return from reg CONCK
         PHA
         LDA CKRTN
         PHA
         PHA            ;Push garbage to account for other pushes done
         PHA            ;in first three bytes of CONCK

         CLC                ;Setup to call CONCK
         LDY    #55.        ;Offset to the address of system CONCK in the
                            ;copy of the original jmp vector.

         LDA    @0E2,Y
         ADC    #3          ;Add 3 so you enter right after the three
                            ;instructions you duplicated at CONCKHDL.
         STA    ROUTINE
         INY
         LDA    @0E2,Y
         ADC    #0
         STA    ROUTINE+1
                            ;In this example we don't have to worry about
                            ;the machine state here as we are restoring
                            ;it after we call CONCK

         JMP    @ROUTINE ;Goto system CONCK and return to CONCKRTN

CONCKRTN PLA                ;Restore state of machine
         TAY
         PLA
         TAX
         PLA
         PLP
         RTS                ;Return to the guy who called CONCK.
-------------------------------------------------------------
```

MAKING ATTACH.DRIVERS

1. Xecute the standard 1.1 LIBRARY program.
2. The output code file should be ATTACH.DRIVERS or could be named somethine else and renamed ATTACH.DRIVERS when you put it on the boot disk.
3. For the Link code file use the code file of your first driver.

4. Copy its slot #1 into slot #0 of ATTACH.DRIVERS.
5. As long as you have more drivers to add, use N(EW to get another Link code file and copy it's slot #1 into slots #2,3,...15 of ATTACH.DRIVERS.
6. When done, type 'Q' then 'N' followed by a RETURN for the notice. See the 1.1 Operating System Reference Manual for further info on the LIBRARY program.

THE WORKINGS OF SYSTEM.ATTACH

If it is on the boot disk, SYSTEM.ATTACH is Xecuted by the operating system (both regular 1.1 and runtime 1.1) before SYSTEM.STARTUP. The 1.1 runtime system will use a runtime version of SYSTEM.ATTACH.

The error messages that can be generated by SYSTEM.ATTACH are:

1. ERROR =>No records in ATTACH.DATA
2. ERROR =>Reading segment dictionary of ATTACH.DRIVERS
3. ERROR =>reading driver
4. ERROR =>A needed driver is not in ATTACH.DRIVERS
5. ERROR =>ATTACH.DATA needed by SYSTEM.ATTACH
6. ERROR =>ATTACH.DRIVERS needed by SYSTEM.ATTACH

If all goes well attaching drivers, SYSTEM.ATTACH will display nothing unusual in the regular boot sequence except for extra disk accesses and anything done in the init calls to any of the attached devices.

II.BIOS

This section explains things in the BIOS area that are extensions and modifications that were added to Apple Pascal version 1.1 that were different or not there at all in Apple Pascal version 1.0 (UCSD version II.1).

1. The disk routines have been modified to handle interrupts (So interrupt driven devices could be attached to 1.1 Pascal) if they are being used. To use interrupts, one would have to attach an interrupt driver, then patch the IRQ vector (FFFE hex) to point to this driver. The Pascal system is defined to come up with interrupts turned off so, once the driver is brought in and the IRQ patched, interrupts must be turned on. The driver's init call could patch the IRQ and turn on interrupts. The disk routines save the current state of the system and turn interrupts off only during crucial time periods, the state of the system is returned during non crucial time periods so interrupts can be handled. This has not been tested at this time, so there is no data concerning the maximum interrupt response time delay.

2. The control word parameter in UNITREAD and UNITWRITE was not passed on to the BIOS level routines from the RSP level. This has been done

in 1.1 to allow the changes to the control word listed below under special character checking and also so user defined units or attached Pascal units can use the user defined bits of the control word.

3. IORESULTS 128-255 are available for user definition on user defined devices.

4. UNITSTATUS has been implemented in the Apple II Pascal 1.1 system. This works for the Pascal system units as described in the ATTACH part of this document. For user defined units, Unitstatus can be used for whatever necessary.

   Unitstatus is a procedure that can be called from the Pascal level in the same way Unitread can. It has three parameters:

   1. unit#.
   2. pointer to a buffer.
      (any size buffer you want of type Packed Array of Char)
   3. control word.

   When you make a Unitstatus call from Pascal, the call should look like:

   ```
   UNITSTATUS(UNITNUM,PAC,CONTROL);
   ```

   Where UNITNUM & CONTROL are integers and PAC is a Packed Array of CHAR or a STRING and may be subscripted to indicate a starting position to transfer data to or from. See further information on what Unitstatus is defined to do for the various devices in the ATTACH part of this document.

   The control word will tell the status procedure for a particular unit what information about the unit you want. Bit 0 of this word should equal 1 for input status and 0 for output status. Unitstatus is implemented with bit 1 of the control word =1 meaning the call is for unit control. When this bit =0 the call is for unitstatus. In all cases bits 2-12 are reserved for system use and bits 13-15 are available for user defined funtions.

   An entry in the jump vector has been made for each of the system Unitstatus calls, i.e. CONSOLESTAT,PRINTERSTAT,REMOTESTAT,etc.. Unitstatus calls to a user defined device (128-143) will all go through the same jump vector location.

5. The handling of CTRL-C by the Apple bios was non standard in 1.0. The UCSD BIOS definition specifies that a CTRL-C coming from REMOTE: or the PRINTER: should be placed in the input buffer and then no more characters should be received. Our bios did fill the buffer with nulls including the place where the CTRL-C was to go. Apple Pascal's BIOS now conforms to the standard definition, where the null filling of the buffer is done only when CTRL-C comes from the CONSOLE: (#1:).

6. The unitio routines can be accessed from assembly procedures by pushing the correct parameters on the stack and using the jump vector to get to the BIOS routine. A seperate document needs to be written describing how this is done and pointing out the problems doing it in the case of the CONSOLE:,SYSTERM:,PRINTER: & REMOTE: units.

These problems are concerned with the special character handling done in the RSP for these units. The assembly procedures calling the pascal drivers for these units would either have to repeat portions of the RSP code themselves or not get the special character handling provided by the RSP. Calling the CONSOLE: init routine requires pointers to syscom and the break routine to be passed on the stack. These pointers are now stored in a fixed location so assembly routines wanting to call coninit can get at them. See the locations section.

7. Suppression of Special Character Checking.

   Special characters in the Pascal system are of three types:

   A. Chars used to control the 40 character screen. These are ctrl-A,Z,W,E & K.

   B. Pascal system control chars for general CONSOLE: use. These are ctrl-S & F.

   C. Types A & B are checked for by the CONCK funtion in the bios. There are other special chars checked for in the RSP. These are ctrl-C, DLE, and CR (line feeds are automatically appended to CR). With UNITREAD and UNITWRITE the automatic handling done by the Pascal system of these characters can be turned off. To turn off DLE expansion and EOF checking give bit 2 of the control word a value of 1. The automatic adding of line feeds to carriage returns can be suppressed by setting bit 3 of the control word to 1.

      A way was needed to suppress special handling for types 'A'&'B'. This can now be done in two ways. First, the control word of UNITR/W will turn off checking for type 'A' control chars if bit 4 is set and will turn off checking for type 'B' chars if bit 5 is set. In this mode, the special char handling will only be turned off during that particular unitio. This will be be done for you in the RSP by setting bits in a byte 'SPCHAR' at location BF1C. The CONCK routine will look at bit 0 of SPCHAR and if set will not look for the type 'A' chars; if bit 1 is set, it will not look for the type 'B' chars. If you set these bits in the SPCHAR yourself instead of letting the RSP do it through the unitio control word, then the associated special character checking will be turned off until you reboot or reset the bits again. When special char checking is turned off, the chars are passed back to the Pascal level like all other chars would be. You can use these added features to redefine the system special chars in a particular application program or to just disable them.

8. The EOF char (ctrl-C) causes a lot of problems in the Pascal system. The cause of the problems is that the editor looks for this character to end many of it's editing modes. The editor has it's own getchar routine which reads each character the user enters from SYSTERM:. When reading from SYSTERM: instead of the CONSOLE:, the EOF char is passed back as any other character but it still ends the current call to unitread. The editor echoes each char to the CONSOLE: itself until it comes to ctrl-C. The operating system and the filer both use the getchar routine in the operating system. This routine is defined to re-init the system if it gets a ctrl-C from the CONSOLE: and it reads from the CONSOLE:, not SYSTERM:. You must be sure not to end responses with control-C except for the cases (in the editor only) that are

supposed to end with control-C. See the 1.1 Operating System Reference Manual.

9. The bios card recognizing section has been enhanced to recognize a new 'FIRMWARE' type card. This card will allow OEM's to have their drivers in their own firmware on the card. Routines have been added to allow for init,read,write & status calls to this new type card. This protocol has been documented and is attached as an appendix to this document.

10. As you can see, the Pascal system memory usage is scattered all over the 64k space. The Apple II was not designed with a stack machine, like the Pascal P-machine, in mind. We don't need any more constraints fixing certain pieces of the system to certain EXACT places. To make the best use of the space we have, we must have the ability to move things around. To achieve this goal, we intend the following:

    A. To stop people from writing things that peek here and poke there and expect things to stay exactly where they were for future versions.

    B. Various people need space for patch areas and other purposes. All programs have to be written so this space does not have to be in a permanent fixed location if this is at all possible. The areas reserved for system use are filling up fast, we need to avoid using them. You can get space dynamically using NEW but you must be careful that this space stays around for the whole time you need it. If you are attaching a driver, you can get buffer space in the driver by using .WORD or .BLOCK in the Assembler. This space can be accessed from outside the driver if you know the offset to the start of this space from the start of the driver. This method could even be used to get space below the heap by attaching a driver to one of the user defined devices that is a large .BLOCK and is only used as a buffer. You can get the address of this buffer (of a driver) from the jump vector that has a pointer to the driver. Pointers to all the jump vectors are in zero page, see the locations section below.

    C. The jump vector will have a fixed order for version 1.1 and future versions. The order is the same as in the old version 1.0 with the new entrys added to the bottom. The setup for the jump vector and getting into the BIOS is different than the old 1.0 system. Here is how the new system is set up with the fixed order for the jump vector:

```
;-----------------------------------------------------
;
;  MAIN BIOS JUMP TABLE CALLED FROM INTERPRETER
;  (FOLLOWED BY REAL JUMP TABLE AT FIXED OFFSET)
;  RSP CALLS COME TO THIS JUMP VECTOR
;
;-----------------------------------------------------
```

```
BIOS            JSR SAVERET           ;CONSOLE READ    ;Jump vector before fold.
                JSR SAVERET           ;CONSOLE WRITE
                JSR SAVERET           ;CONSOLE INIT
                JSR SAVERET           ,PRINTER WRITE
                JSR SAVERET           ;PRINTER INIT
                JSR SAVERET           ;DISK WRITE
                JSR SAVERET           ;DISK READ
                JSR SAVERET           ;DISK INIT
                JSR SAVERET           ;REMOTE READ
                JSR SAVERET           ;REMOTE WRITE
                JSR SAVERET           ;REMOTE INIT
                JSR SAVERET           ;GRAFIC WRITE
                JSR SAVERET           ;GRAFIC INIT
                JSR SAVERET           ;PRINTER READ
                JSR SAVERET           ;CONSOLE STAT
                JSR SAVERET           ;PRINTER STAT
                JSR SAVERET           ;DISK STAT
                JSR SAVERET           ;REMOTE STAT
KCONCK          JSR SAVERET           ;To get to CONCK from CONCKVEC
                JSR SAVERET           ;USER READ    For UDRWIS
                                      ;USER WRITE
                                      ;USER INIT
                                      ;USER STAT
                 JSR SAVERET          ;For PSUBDR
                 JSR SAVERET          ;IDSEARCH
                     .
                     .
                     .

;--------------------------------------
;
;  THIS JUMP TABLE MUST BE OFFSET
;  FROM BIOSTBL BY EXACTLY $5C.
;  SYSTEM.ATTACH MODIFYS THIS JUMP
;  VECTOR TO POINT TO ATTACHED DRIVERS
;  FOR THE STANDARD SYSTEM UNITS.
;
;--------------------------------------

BIOSAF          JMP CREAD             ;Jump vector after fold.
                JMP CWRITE
                JMP CINIT
                JMP PWRITE
                JMP PINIT
                JMP DWRITE
                JMP DREAD
                JMP DINIT
                JMP RREAD
                JMP RWRITE
                JMP RINIT
                JMP IORTS            ;Do nothing for GRAFWRITE.
                JMP GRAFINIT
                JMP IORTS            ;Do nothing for PRINTER: read.
                JMP CSTAT
                JMP ZEROSTAT         ;For PRINTER: stat, pop params & store 0
                                     ;in 1st buffer word.
                JMP DSTATT
                JMP ZEROSTAT         ;For REMOTE: stat, pop params & store 0
```

```
                             ;in 1st buffer word.
         JMP CONCK
         JMP UDRWIS          ;Routine to get to user defined devices, see
                             ;ATTACH part of document for description of
                             ;this routine.
         JMP PSUBDR          ;Routine to get to drivers that are substituted
                             ;for the standard Pascal disk units 4,5,9..12.
                             ;See ATTACH part of document for description of
                             ;this routine.
         JMP IDS


;----------------------------------------
;
; STRIP LOCAL RETURN ADDR,
; STRIP PASCAL ADDR AND SAVE IN RETL,RETH
; PLACE 'GOBACK' ON RETURN STACK
; THEN RESTORE LOCAL RET ADDR & RETURN
; MEANWHILE UNFOLD BIOS INTO DXXX
;
;----------------------------------------
SAVERET   STA TT1            ;SAVE A REG
          PLA
          CLC
          ADC #05A           ;ADD OFFSET TO JUMP TABLE (BIOSAF)
          STA TT2            ;LOCAL RET ADDR
          PLA
          ADC #0
          STA TT3
          PLA
          STA RETL           ;PRESERVE PASCAL RETURN
          PLA
          STA RETH
          .IF RUNTIME=0
          LDA 0C083          ;UNFOLD BIOS INTO DXXX
          .ENDC
          LDA TT1            ;RESTORE A-REG
          JSR SAVRET2        ;PUTS 'GOBACK' ON STACK

;----------------------------------------
;
; FOLD INTERP INTO DXXX
; THEN RETURN TO PASCAL VIA
; RETURN ADDR SAVED IN RETL,RETH
;
;----------------------------------------
GOBACK    STA TT1            ;SAVE A-REG
          LDA RETH
          PHA
          LDA RETL
          PHA
          .IF RUNTIME=0
          LDA 0C088          ;FOLD INTERP INTO DXXX
          .ENDC
          LDA TT1
          RTS                ;AND BACK TO PASCAL

SAVRET2   JMP @TT2           ;JUMP INTO JUMP TABLE (BIOSAF)
```

D. In zero page are two words pointing to the base of the two jump vectors (before and after the fold). These are stored in PERMANENT locations that had a value of 0 in the old 1.0 release and were not used by the system (see locations section). Applications needing to patch the jump vectors can store the offset from the vector base in the Y reg and use indirect indexed addressing to do the patch. The application will need to have the vector base locations for the old release hardcoded in as the base pointer for the old 1.0 release will be 0. If you want to write an application that works with 1.0 and 1.1 and future versions, you know if the zero page vector pointers are 0 it's the 1.0 system otherwise it's 1.1 or a future version which will use the same protocols as 1.1 as described in this document.

   It is important that any application patching the jump vector temporarily then returning it to its original value get the original value from the vector itself before the patch and put it in a storage location. When the vector needs to be restored to it's original state, use this storage location for it's original value. The patches should be done in this manner so the applications doing the patches will always return the system to it's original state no matter what past, present or future Pascal version it is patching.

E. For CONSOLE: init to be used from assembly routines the locations of SYSCOM and the BREAK routine have to be available. The CONINIT routine requires these on the stack. Pointers to SYSCOM and BREAK will be stored by the interpreter boot in a PERMANENT location in the BF00 page (see locations section).

F. Since the old 1.0 release, the code to jump to the CONCK routine has been set up at location BF0A. Anyone wishing to get to the CONCK routine should do a JSR BF0A as this will always get them there no matter where the CONCK routine really is. The keypress function has been changed to conform to this new convention but it will use the old convention if it is working from within an old system. Do not try to get to CONCK in this way from within an ATTACHED driver as you will loose your return address to Pascal. See ATTACH part of this document for how to get to CONCK from an attached driver.

G. There is now a version byte so one can tell which version (1.0, 1.1, etc.) of Apple Pascal he is working with. There is also a flavor byte to tell one which flavor of this version he has (regular, runtime, runtime without sets, etc.). (see locations section)

11. Whenever SYSTEM.ATTACH is used, it will make a copy of the original BIOS jump vector (the after fold vector that has the actual driver addresses in it) and put this below the heap with the drivers that are attached. It will leave a pointer to this copy of the vector at location 00E2. You can use this vector in you drivers to get to the standard Apple drivers for any device. This way you can define a driver that does something above and

beyond the standard Apple driver yet this new driver can still make use of the standard Apple driver. See the ATTACH part of this document for more information.

12. In the RSP are two vectors that tell the RSP what is legal (input &-or output) for a particular character orientated device (CONSOLE:, REMOTE: & PRINTER:). For example it tells the RSP that it is illegal to read from the PRINTER:. If you wanted to ATTACH a PRINTER: driver so you could read from the PRINTER:, you would have to change this vector. 00E4 points to the READTBL vector and 00E6 to the WRITTBL vector. Let's take the READTBL for an example:

```
READTBL         ;table of routine addresses to be called when
                ;writing to that unit (disk I/O does not use
                ;this table).
                ;an entry=0 means that the operation is illegal
                ;for that unit.
                .WORD     BIOS+CONREAD    ;unit 1
                .WORD     BIOS+CONREAD    ;unit 2
                .WORD     0               ;unit 3
                .WORD     0               ;4 & 5 are disk units
                .WORD     0
                .WORD     0               ;6 is PRINTER:
                .WORD     BIOS+REMREAD    ;unit 7
                .WORD     0               ;8 is rem write which has
                                          ;an address in the WRITTBL
```

Here BIOS refers to the base of the jump vector before the fold and CONREAD is the offset off the base of that vector to get to the jump to the CONSOLE: read routine (for CONSOLE: read the offset is 0, for CONSOLE: write it's 3, etc). The value for BIOS is the pointer stored in location 00EC mentioned in the locations section below.

LOCATIONS.

These are the locations of new system permanents mentioned in this document, all pointers are set up by the system and are stored low byte first. Do not modify what is stored in these pointers (except for SPCHAR if you want to suppress special character checking) since the system uses this information too. These locations are defined to have the same function & remain in the same place for future versions of Apple II Pascal.

BF1C	SPCHAR	(To control special chars)
BF1D	IBREAK	(Set by boot in interp for assembly calls to CONINIT)
BF1F	ISYSCOM	('')
BF21	VERSION	(1 byte Version # of system, =2 for the new release, 0 for the old 1.0 release)
BF22	FLAVOR	(This byte tells which flavor [runtime,regular, etc.] of this VERSION you are dealing with) The encoding is: 1 -->regular system

runtime versions:
2 -->LC-ALL (LC- means no language card)
3 -->LC-no sets
4 -->LC-no floating point
5 -->LC-no sets or floating point
6 -->LC+ALL
7 -->LC+no sets
8 -->LC+no floating point
9 -->LC+no sets or floating point

This flavor byte is 0 in the old 1.0 release.

BFC0-BFFF	BDEVBUF	(Area for non Apple boot devices, like the CORVUS)
00E2	ACJVAFLD	(Pointer to ATTACH copy of the original Jump Vector after the fold)
00E4	RTPTR	(Pointer to READTBL)
00E6	WTPTR	(Pointer to WRITTBL)
00E8	UDJVP	(Pointer to user device jump vector)
00EA	DISKNUMP	(Pointer to disknum vector)
00EC	JVBFOLD	(Pointer to jump vector before fold)
00EE	JVAFOLD	(Pointer to jump vector after fold)
FFF6		(Version word which = 1 for version 1.0 and = 0 for version 1.1 This version word should not be used at runtime to tell which version you have. For that use the version byte mentioned above. This word should only be used by software that wants to see which SYSTEM.APPLE it is dealing with by looking at the contents of this word in the SYSTEM.APPLE file when it is not loaded in memory)
FFF8		(Start vector)
FFFA		(NMI non maskable interrupt vector)
FFFC		(RESET vector)
FFFE		(IRQ interrupt request vector)

The locations and code in the 1.0 'PRELIMINARY APPLE PASCAL GUIDE TO INTERFACING FOREIGN HARDWARE' BIOS document are not the same for Apple Pascal 1.1 and that document clearly stated we would not commit ourselves to keeping them the same.

Pascal 1.1 Firmware Card Protocol

One major problem with Apple Pascal 1.0 is the way it deals with peripheral cards. It was set up to work with the four peripheral cards that Apple supported at the time of its release (the disk,communciations,serial and parallel cards) and had no mechanism for interfacing any other devices. Since Apple as well as many other vendors continue to produce new peripherals for the Apple][, a new protocol was designed and implemented in the Pascal 1.1 BIOS which allows new peripheral cards to be introduced to the system in a consistent and transparent fashion. The new protocol is called the "firmware card" protocol since the BIOS deals with these cards by making calls to their firmware at entry points defined by a branch table on the card

itself. The new protocol fully supports the Pascal typeahead function and KEYPRESS will work with firmware cards used as CONSOLE devices. The following paragraphs describe the firmware card protocol in full detail.

A firmware card may be uniquely identified by a four byte sequence in the card's $CN00 ROM space. Location $CN05 must contain the value $38 and location $CN07 must contain $18. Note that these are identical to the Apple Serial Card. A firmware card is distinguished from a serial card by the further requirement that location $CN0B must contain the value $01. This value is called the "generic signature" since it is common to all firmware cards. The value at the next sequential location, $CN0C, is called the "device signature" since it uniquely identifies the device.

The device signature byte is encoded in a meaningful way. The high order 4 bits specify the class of the device while the low order four bits contain a unique number to distinguish between specific devices of the same class. The appendix to this document defines some device class numbers; in any case vendors should contact Apple Technical Support to make sure they use a unique number for their device signature. Although the device signature is ignored by the 1.1 BIOS, it may be used by applications programs to identify specific devices.

Following the 2 signature bytes is a list of four entry point offsets starting at address $CN0D. These four entry points must be supported by all firmware cards. They are the initialization, read, write and status calls. The BIOS takes care of disabling the $C800 ROM space of all other cards before calling the firmware routines.

The offset to the initialization routine is at location $CN0D. Thus, if $CN0D contains XX, the BIOS will call $CNXX to initialize the card. On entry, the X register contains $CN (where N is the slot number) and the Y register contains $N0. On exit, the X register should contain an error code, which should be 0 if there was no error. This error code is passed on to the higher levels of the system in the global variable "IORESULT". Registers do not have to be preserved.

The offset to the read routine is at location $CN0E. On entry, the X register will contain $CN and the Y register will contain $N0. On exit, the A register should contain the character that was read while the X register contains the IORESULT error code. The A and Y registers do not have to be preserved.

The offset to the write routine is at location $CN0F. On entry, the A register contains the character to be written while the X register contains $CN and the Y register contains $N0. On exit the X register should contain the IORESULT error code (which should be 0 for no error). The A and Y registers do not have to be preserved.

The offset to the status routine is at location $CN10. On entry, the X register contains $CN and the Y register contains $N0 while the A register contains a request code. If the A register contains 0, the request is "are you ready to accept output?". If the A register contains 1, the request is "do you have input ready for me?". On exit, the driver returns the IORESULT error code in the X register and the results of the status request in the carry bit. The carry clear means "false" (i.e., no, I don't have any input for you), while the carry set means true. Note that the status call must preserve the Y register but does not have to preserve the A register.

Thus, sample code for the first few bytes of a firmware card's $CN00 space should look something like:

```
BASICINIT       BIT     $FF58           ;set the v-flag
                BVS     BASICENTRY      ;always taken
IENTRY          SEC                     ;BASIC input entry point
                DFB     $90             ;opcode for BCC
OENTRY          CLC                     ;BASIC output entry point
                CLV
                BVC     BASICENTRY      ;Always taken
;
;  Here is the Pascal 1.1 Firmware Card Protocol Table
;
                DFB     $01             ;Generic signature byte
                DFB     $41             ;Device signature bye
;
PASCALINIT      DFB     >PINIT          ; > means low order byte
PASCALREAD      DFB     >PREAD          ;offset to read
PASCALWRITE     DFB     >PWRITE         ;offset to write
PASCALSTATUS    DFB     >PSTATUS        ;offset to status routine
```

The above code fulfils all the requirements for both the BASIC and Pascal 1.1 I/O protocols. The routines PINIT, PREAD, etc, are probably jumps into the card's $C800 space which is already properly enabled by the BIOS. The reason the $CN00 space was chosen for the protocol (as opposed to the $C800 space) is that the BASIC protocol requires that all cards have $CN00 ROM space while some smaller cards may not need any $C800 ROM space.

The firware card protocol includes 2 optional calls that do not have to be implemented but would be kind of nice. The BIOS checks location $CN11 to determine if the optional calls are present; if that location contains a $00 then the BIOS thinks the calls are implemented. Thus if your card does not implement the optional calls, you should ensure that $CN11 contains a non-zero value. The two optional calls are a control call pointed to by $CN12 and an interrupt handler call pointed to by $CN13.

The control call entry point is specified by the offset at $CN12. On entry, the X register contains $CN, the Y register contains $N0 and the A register contains the control request code. Control requests are defined by the device. On exit the X register should contain the IORESULT error code.

The interrupt poll entry point is specified by the offset at $CN13. On entry, the X register contains $CN and the Y register contains $N0. The interrupt poll routine should poll the card's hardware to determine if it has a pending interrupt; if it does not it should return with the carry clear. If it does, it should handle the interrupt (including disabling it) and return with the carry set. Also, the X register should contain the IORESULT error code which should be 0 if there was no error. An interrupt polling routine must be careful not to clobber any zero page or screen space temporaries.

The control and interrupt requests are not implemented in the Pascal 1.1 BIOS but it would be nice to support them if possible as they may be implemented in later versions of the Pascal BIOS as well as other forthcoming operating system environments for the Apple][.

Note that the firmware card signature is a superset of the Apple serial card signature as recognized by the Pascal 1.0 BIOS. This allows a firmware card to function with both Pascal 1.0 and Pascal 1.1. If a card wishes to work with Pascal 1.0 as a "fake" seral card, it must provide an input entry point at $C84D and an output entry point at $C9AA. Note that since Pascal 1.0 will think the card is a serial card, typeahead and KEYPRESS capabilities will be lost.

Additional Notes

1. The Pascal RSP expects the high order bit of every ASCII character it receives from the Console read routine to be clear. The RSP will not do this for you; you must ensure the high bit of all text your card passes to the RSP from the console read routine is clear.

2. Zero page locations $00 to $35 may be used as temporaries by your firmware, as are the slot 0 screen space locations ($478,$4F8, etc.). In general, peripheral card firmware should be as conservative as possible in their memory usage, preserving zero page contents whenever possible. An interrupt polling routine must not destroy these or any other memory locations.

3. Location $7F8 must be set up to contain the value $CN, where N is the slot number, if your card utilizes the $C800 expansion ROM space. The BIOS does not do this for you; his must be done if you want your card to function in an interrupting environment.

4. The firmware card status routine should be as quick as possible, as it may be called from within the I/O polling loops of many other peripherals if your card is being used as the console device. In no case should the status routine take longer than 100 milliseconds.

5. A firmware card in slot 1 is automatically recognized as the volume "PRINTER:". A firmware card in slot 2 is automatically recognized as the volumes "REMIN:" and "REMOUT:". A firmware card in slot 3 is automactically recognized as the volumes "CONSOLE:" and "SYSTERM:".

APPENDIX

The following numbers correspond to device classes used in the device signature code. Make sure you contact Apple Technical Support to ensure that you have a unique device signature code.

0 -- reserved
1 -- printer
2 -- joystick or other X-Y input device
3 -- I/O serial or parallel card
4 -- modem
5 -- sound or speech device
6 -- clock
7 -- mass storage device

8 -- 80 column card
9 -- Network or bus interface
10 -- Special purpose (none of the above)

11 through 15 are reserved for future expansion

Additional Information

1. The type ahead buffer is located at $03B1 hex and is $4E hex in length. It is implemented with a read pointer (RPTR at BF18 hex) and a write pointer (WPTR at $BF19 hex). At CONSOLE: init time, these should both be set to 0. When a character is detected by CONCK, the WPTR is incremented then compared with $4E. If it is equal to $4E, it is set to $0 (this is a circular buffer). Then the WPTR is compared with RPTR and if they are equal the buffer is full. If the buffer is not full, the character is stored at $03B1+the value in WPTR.

   When removing a character from the type ahead buffer, use the following sequence. Compare the RPTR with WPTR and if they are equal, the buffer is empty and you must wait until a character is available from the keyboard. If they are not equal, increment the RPTR and compair it to $4E. If it equals $4E, set it to $0. Now get the character from location $03B1+the value in RPTR.

   If you are implementing your own type ahead, you can do it however you wish. This information is made available in case you want to check for input from another device as well as the standard system CONSOLE: and have characters from that device be put in the system type ahead buffer.

2. The example drivers in this document did not show the setting of the IORESULT in the X register. This would be done in the code specific to your driver and should allways be set to something (0 if there are no errors). If there are errors, set it as described elsewhere in this document and the Pascal Manuals.

3. For further information, see the newest edition of the Apple II Reference Manual.

4. These listings from the BIOS are included to show you how we implemented certain system drivers. You cannot rely on the locations of these to stay in the same place in the BIOS in future releases of Apple II Pascal nor can you rely on the routines themselves staying the same. They are only included as examples and to give you information that may not be documented elsewhere. This is not a complete BIOS listing so you may find references to routines or locations that are not included in this listing. The only locations that will be sure to remain the same for future releases are those mentioned in the LOCATIONS section above. We are against you poking the BIOS yourself to change or overwrite any of these routines. We did not include this information so you could poke the BIOS. If you do modify the BIOS, it is completely at your own risk! We have provided the ATTACH utility so you can add your own drivers the system without poking the BIOS and this is the way it should be done! If you have special requirements that are not solved by ATTACH, please

contact Apple Technical Support.

```
;-------------------------------------
;
; ZERO PAGE PERMANENTS
;
;-------------------------------------
FIRST        .EQU 0F0          ;START ZERO PAGE USE
BAS1L        .EQU FIRST        ;SCREEN 1 PTR
BAS1H        .EQU FIRST+1
BAS2L        .EQU FIRST+2      ;SCREEN 2 PTR
BAS2H        .EQU FIRST+3
CH           .EQU FIRST+4      ;HORIZ CURSOR, 0..79
CV           .EQU FIRST+5      ;VERT CURSOR, 0..23
TEMP1        .EQU FIRST+6
TEMP2        .EQU FIRST+7
SYSCOM       .EQU FIRST+8      ;2 BYTES PTR TO SYSCOM AREA

;-------------------------------------
;
; BF00 PAGE PERMANENTS
;
;-------------------------------------
CONCKVECTOR .EQU 0BF0A         ;4 BYTES
SCRMODE      .EQU 0BF0E
LFFLAG       .EQU 0BF0F
NLEFT        .EQU 0BF11
ESCNT        .EQU 0BF12
RANDL        .EQU 0BF13
RANDH        .EQU 0BF14
CONFLGS      .EQU 0BF15
BREAK        .EQU 0BF16        ;2 BYTES
RPTR         .EQU 0BF18        ;1 BYTE
WPTR         .EQU 0BF19        ;1 BYTE
RETL         .EQU 0BF1A
RETH         .EQU 0BF1B
SPCHAR       .EQU 0BF1C        ;00 MEANS DO ALL SPECIAL CHARACTER CHECKING
                               ;01 MEANS DON'T CHECK FOR APPLE SCREEN STUFF
                               ;02 MEANS DON'T CHECK FOR OTHER SCREEN STUFF
IBREAK       .EQU 0BF1D        ;INTERP STORES BREAK & SYSCOM ADR HERE FOR
ISYSCOM      .EQU 0BF1F        ;USER ROUTINES TO GET AT
VERSION      .EQU 0BF21        ;VERSION OF SYSTEM SET TO 2 FOR APPLE 1.1
FLAVOR       .EQU 0BF22        ;SEE TABLE IN INTERP BOOT
SLTTYPS      .EQU 0BF27        ;BF27..0BF2E
XITLOC       .EQU 0BF2F        ;INTERP INITS THIS TO LOCATION OF XIT
             ;FORTRAN PROTECTION USES BF56..BF7F
             ;VENDOR BOOT DEVICES CAN USE BFC0..BFFF

;-------------------------------------
;
; MISCELANEOUS PROGRAM EQUATES
;
;-------------------------------------
BUFFER       .EQU 0200         ;TEMP HSHIFT BUFFER (OVERLAPS DISK BUF)
CONBUF       .EQU 03B1         ;78 CHAR TYPE-AHEAD BUF
CBUFLEN      .EQU 04E          ;78 DECIMAL
NCTRLS       .EQU 14.          ;# CTRL CHARS IN TABLE
SIGVALUE     .EQU 1
```

```
BYTEPSEC    .EQU 256.           ;DISK INFO FOR DISKSTAT
SECPTRAK    .EQU 16.
TRAKPDSK    .EQU 35.
UDJVP       .EQU 0E8            ;0 PAGE JUMP VECTOR POINTER LOCATIONS
DISKNUMP    .EQU 0EA
JVBFOLD     .EQU 0EC
JVAFOLD     .EQU 0EE
HCMODE      .EQU 0E1            ;THESE TWO BYTES USED FOR HIRES STUFF
HSMODE      .EQU 0E0


JVECTRS     .WORD       UDJMPVEC
            .WORD       DISKNUM
            .WORD       BIOS
            .WORD       BIOSAF

;--------------------------------------
;
; HARD RESET INITIALIZATION
;
;--------------------------------------
START       CLD                 ;SET HEX MODE
            SEI                 ;MAKE SURE INTERRUPTS ARE OFF.

;--------------------------------------
;
; CLEAR ALL MEMORY 0 TO BFFF
; (RUN-TIME SYSTEM:0 TO TOPMEM + BF PAGE);
;
;--------------------------------------
            LDA #0
            STA ZEROL
            STA ZEROH
            TAY
            TAX
ZERLP       STA (ZEROL),Y       ;WRITE A BYTE OF 0
            INY                 ;BUMP POINTER
            BNE ZERLP           ;LOOP TILL NEXT PAGE
            INC ZEROH           ;BUMP MSB POINTER
            INX
            .IF RUNTIME=1
            CPX #TOPMEM         ;DONE CLEARING MEM?
            BNE $1
            LDX #0BF            ;CLEAR BF PAGE
            STX ZEROH
$1:         CPX #0C0
            BNE ZERLP
            .ELSE
            CPX #0C0            ;DONE CLEARING BFXX?
            BNE ZERLP
            .ENDC

;----------------------
;
; CHECKSUM PROMS ON EACH SLOT
; TO FIND OUT WHO'S OUT THERE
;
; SUM TWICE TO TELL IF CARD THERE
```

```
; IF SUMS DONT MATCH THEN NO PROM IS THERE
; IF MS BYTE OF SUM=0 THEN NO PROM IS PRESENT
;
;----------------------
            LDY #0C7            ;POINT TO SLOT 7 PROM
NXTCRD      STY CKPTRH          ;(CKPTRL=0 FROM MEM CLEAR)
            JSR CKPAGE          ;16 BIT SUM IN X,A
            STA CHECKL
            STX CHECKH          ;SAVE FOR MATCH
            JSR CXPAGE          ;SUM AGAIN
            CPX #0              ;WAS MSB ZERO?
            BEQ NOPROM          ;YES NO PROM ON CARD
            CMP CHECKL          ;LSB MATCH?
            BNE NOPROM          ;NO, NO PROM ON CARD
            CPX CHECKH
            BNE NOPROM          ;MSB DIDNT MATCH
            BEQ SKIPIORTS       ;ALWAYS TAKEN

;---------------------------
;
; TABLE OF CN05 AND CN07 BYTES OF EACH CARD
;
;---------------------------
CN05BYTS    .BYTE 003,018,038,048
CN07BYTS    .BYTE 03C,038,018,048

;---------------------------
;
; NOW THAT WE KNOW A CARD IS THERE,
; EXAMINE GN05 AND CN07 BYTE TO
; DETERMINE WHICH CARD IT IS
;
; SET CARDTYPE AS FOLLOWS:
; 0=CKSUM NOT REPEATABLE OR MSB=0
; 1=CKSUM REPEATABLE,CARD NOT RECOGNIZED
; 2=DISK CARD (BYTE 07= 03C)
; 3=COM CARD  (BYTE 07= 038)
; 4=SERIAL    (BYTE 07= 018)
; 5=PRINTER   (BYTE 07= 048)
; 6=FIRMWARE  (BYTE 07= 048)
;----------------------------
SKIPIORTS   LDX #5              ;4 TYPES OF CARDS
NXTYP       LDY #5              ;CHECK BYTE CN05 OF CARD
            LDA (CKPTRL),Y
            CMP CN05BYTS-2,X    ;MATCH TABLE?
            BNE TRYNXT          ;NO, TRY NEXT IN LIST
            LDY #7
            LDA (CKPTRL),Y      ;TEST CN07 BYTE
            CMP CN07BYTS-2,X    ;MATCH TABLE?
            BEQ STOR            ;BOTH MATCHED, CARD RECOGNIZED
TRYNXT      DEX                 ;BUMP TO NEXT IN LIST
            CPX #2              ;TRY ALL TYPES IN LIST
            BCS NXTYP           ;IF NOT IN LIST,FALL THRU WITH X=1
STOR        CPX #4              ;IS IT A SERIAL CARD?
            BNE STOR1
            LDY #0B
            LDA (CKPTRL),Y
            CMP #SIGVALUE
```

```
            BNE STOR1
            LDX #6
STOR1       LDY CKPTRH
            TXA
            STA SLTTYPS-0C0,Y
NOPROM      LDY CKPTRH
            DEY                 ;BUMP TO NEXT LOWER SLOT
            CPY #0C0            ;SLOTS 7 DOWNTO 1 DONE?
            BNE NXTCRD          ;LOOP TILL 7 SLOTS DONE
                                ;LEAVE WITH Areg:=0
;--------------------------------------
;
; SET UP CONCK VECTOR FOR KEYPRESS FUNCTION
;
;--------------------------------------
            BEQ $2              ;ALWAYS BRANCHES
$1          JSR KCONCK          ;HERE ARE THE 2 INSTRUCTIONS TO BE TRANSFERRED
            RTS
$2          LDY #3              ;TRANSFER 4 BYTES TO BF0A
$21         LDA $1,Y
            STA CONCKVECTOR,Y
            DEY
            BPL $21

            ;SET UP JUMP VECTOR POINTERS IN 0 PAGE
            LDY #7
$3          LDA JVECTRS,Y
            STA UDJVP,Y
            DEY
            BPL $3

;--------------------------------------
;
; SET SCREEN MODE ETC
;
;--------------------------------------
            LDA #80
            STA HCMODE
            LDA 0C051           ;SET TEXT MODE
            LDA 0C052           ;SET BOTTOM 4 GRAFIX
            LDA 0C054           ;SELECT PRIMARY PAGE
            LDA 0C057           ;SELECT HIRES GRAFIX
            LDA 0C010           ;CLEAR KEYBOARD STROBE
            JSR FORM            ;ERASE SCREEN
            JSR INVERT          ;PUT CURSOR ON SCREEN
            JSR DRESET          ;DO ONCE ONLY DISK INIT
            LDA SLTTYPS+3       ;WHAT CARD IN SLOT 3?
            LDY #030            ;SLOT 3
            JSR GENIT           ;SET BAUD IF COM OR SER THERE
            CPX #0              ;WAS AN EXTERNAL CONSOLE THERE?
            BNE STARTUP         ;NO,USE APPLE SCREEN
            LDA #4
            STA SCRMODE         ;SET BIT 2 FOR EXT CON
STARTUP     JMP JPASCAL         ;FOLD IN INTERP AND START PASCAL

;------------------------
;
; SUB TO CHECKSUM ONE PAGE
```

```
;
CKPAGE      LDA #0
            TAX                     ;CLEAR SUM
            TAY                     ;CLEAR INDEX
CKNX        CLC
            ADC (CKPTRL),Y          ;ADD BYTE
            BCC NOCRY
            INX                     ;INC HI BYTE IF CARRY
NOCRY       INY                     ;BUMP INDEX
            BNE CKNX                ;SUM 256 BYTES
            RTS                     ;RETURN SUM IN X,A AND Y=0

;--------------------------------------------------------------
;
;  BIOS HANDLERS FOR LOGICAL AND PHYSICAL DEVICES.
;
;--------------------------------------------------------------




;-------------------------------------
;
; CONSOLE CHECK FOR CHAR AVAIL
; STATUS AND ALL REGS PRESERVED
; IF CHAR AVAIL,PUT IN CONBUF AND INC WPTR.
;
; WARNING...THIS ROUTINE ALSO CALLED FROM DISK ROUTINES
;
;-------------------------------------
CONCK       PHP
            PHA
            TXA
            PHA
            TYA
            PHA
RNDINC      INC RANDL.              ;BUMP 16 BIT RANDOM SEED
            BNE RNDOK
            INC RANDH
RNDOK       LDA SLTTYPS+3           ;WHAT CARD IS IN SLOT 3?
            CMP #3                  ;IS IT A COM CARD?
            BEQ COMCK               ;YES,GO CHECK IT
            CMP #4                  ;IS IT A SERIAL CARD?
            BEQ JDONCK              ;YES,IT CANT BE TESTED
            CMP #6
            BEQ FIRMCK
TSTKBD      LDA 0C000               ;TEST APPLE KEYBOARD
            BPL JDONCK              ;NO CHAR AVAIL
            STA 0C010               ;CLEAR KEYBD STROBE
            AND #07F                ;MASK OFF TOP BIT
            TAX                     ;See if checking for apple special chars is
            LDA SPCHAR              ;turned off.
            ROR A
            BCS NOTFOLP2            ;Jump if so
            TXA
            CMP #11.                ;CTRL-K?
```

```
                BNE NOTK
                LDA #05B                ;YES,REPLACE WITH LEFT SQR BRACKETT
NOTK            CMP #1                  ;CTRL-A?
                BNE NTTAB
                JSR HTAB                ;YES,TAB NEXT MULT 40
                LDA CONFLGS
                AND #0FE
                STA CONFLGS             ;CLEAR AUTO-FOLLOW BIT
                JMP DONECK
NTTAB           CMP #26.                ;CTRL-Z?
                BNE NOTFOL              ;NO,PUT CHAR IN BUFFER
                LDA CONFLGS
                ORA #1
                STA CONFLGS             ;SET AUTO-FOLLOW BIT
                BNE DONECK              ;BR ALWAYS


COMCK           LDA 0C0BE               ;CHAR AVAIL?
                LSR A
                BCC DONECK              ;NO CHAR AVAIL
                LDA 0C0BF               ;GET CHAR FROM UART
GOTCHAR         AND #07F                ;MASK OFF BIT 7
NOTFOL          TAX
                LDA SPCHAR              ;See if console special char checking is
                                        ;turned off.
                ROR A
NOTFOLP2        ROR A
                BCS NFMI1               ;Jump if so
                TXA
                LDY #055
                CMP (SYSCOM),Y          ;STOP CHAR?
                BNE NOTSTOP
                LDA CONFLGS
                EOR #080
                STA CONFLGS             ;YES,TOGGLE STOP BIT (BIT 7)
JDONCK          JMP DONECK

FIRMCK          LDA #1
                LDY #030
                JSR FIRMSTATUS
                BCC DONECK
                JSR FREAD1
                JMP GOTCHAR

NOTSTOP         DEY
                CMP (SYSCOM),Y
                BNE NOTBRK
                LDA CONFLGS
                AND #03F
                STA CONFLGS             ;CLEAR FLUSH&STOP BITS
                .IF RUNTIME=0
                 JMP TOBREAK
                .ELSE
                 JMP @BREAK              ;BREAK OUT
                .ENDC
NOTBRK          DEY
                CMP (SYSCOM),Y          ;FLUSH?
                BNE NOTFLUS
```

```
            LDA CONFLGS
            EOR #040
            STA CONFLGS         ;TOGGLE FLUSH BIT (BIT 6)
            JMP DONECK

NFMI1       TXA
NOTFLUS     LDX WPTR
            JSR BUMP
            CPX RPTR            ;BUFFER FULL?
            BNE BUFOK
            JSR BELL
            JMP DONECK          ;BEEP&IGNORE CHAR
BUFOK       STX WPTR
            STA CONBUF,X        ;PUT CHAR IN BUFFER
DONECK      BIT CONFLGS         ;IS STOP FLAG SET?
            BPL CKEXIT
            JMP RNDINC          ;LOOP IF IN STOP MODE

CKEXIT      PLA
            TAY
            PLA
            TAX
            PLA
            PLP
            RTS                 ;ELSE RESTORE STAT AND ALL REG AND RETURN
BUMP        INX                 ;BUMP BUFFER POINTER WITH WRAP-AROUND
            CPX #CBUFLEN
            BNE BMPRTS
            LDX #0
BMPRTS      RTS


;----------------------------------
;
; INITIALIZE CONSOLE:
;
;----------------------------------
CINIT       PLA
            STA TEMP1           ;SAVE RETURN ADDR
            PLA
            STA TEMP2
            PLA
            STA SYSCOM          ;SAVE PTR TO SYSCOM AREA
            PLA
            STA SYSCOM+1
            PLA
            STA BREAK           ;SAVE BREAK ADDRESS
            PLA
            STA BREAK+1
            LDA TEMP2
            PHA                 ;RESTORE RETURN ADDR
            LDA TEMP1
            PHA
            LDA RPTR            ;FLUSH TYPE-AHEAD BUFFER
            STA WPTR
            LDA CONFLGS
            AND #03E
```

```
            STA CONFLGS         ;CLEAR STOP,FLUSH,AUTO-FOLLOW BITS
            JSR TAB3            ;NO,HORIZ SHIFT FULL LEFT
CINIT2      LDX #0              ;CLEAR IORESULT
            RTS                 ;AND RETURN

;----------------------------------
;
; READ FROM CONSOLE:
; KEYBOARD,COM OR SERIAL CARD IN SLOT 3
;
;----------------------------------
CREAD       JSR ADJUST          ;HORIZ SCROLL IF NECESSARY
            LDY #030            ;SLOT 3
            LDA SLTTYPS+3       ;WHAT TYPE OF CARD?
            CMP #4              ;IS IT A SERIAL CARD?
            BNE CREAD2          ;NO,CONTINUE
            JSR RSER            ;YES, READ IT
            AND #7F             ;MASK OFF TOP BIT
            RTS
CREAD2      JSR CONCK           ;TEST CHAR
            LDX RPTR
            CPX WPTR
            BEQ CREAD           ;LOOP TILL SOMETHING IN BUFFER
            JSR BUMP
            STX RPTR            ;BUMP READ POINTER
            LDA CONBUF,X        ;GET CHAR FROM BUFFER
            LDX #0              ;CLEAR IORESULT
            RTS                 ;AND RETURN TO PASCAL

;----------------------------------
;
; INITIALIZE PRINTER:
; PRINTER IS ALWAYS IN SLOT 1
; IT MAY BE A PRINTER,COM,OR SERIAL CARD
;
;----------------------------------
PINIT       LDY #010            ;SLOT 1 ; 010
            LDA SLTTYPS+1       ;WHAT CARD IN SLOT 1?
            CMP #5              ;PRINTER CARD?
            BEQ CLRIO1          ;YES,NO INIT NEEDED
GENIT       CMP #4              ;SERIAL CARD?
            BEQ ISER            ;YES, INIT SER CARD
            CMP #3              ;COM CARD?
            BEQ ICOM            ;YES,INIT COM CARD
            CMP #6
            BEQ FIRMINIT
            LDX #9              ;NONE OF ABOVE,OFFLINE
            RTS

FIRMINIT    PHA
            JSR SER1
            LDY #0D
FVEC1       LDA (TEMP1),Y
            STA TEMP1
            LDY 6F8
            PLA
            JMP @TEMP1
```

- --

```
;--------------------------------
;
; INITIALIZE REMOTE:
; REMOTE IS ALWAYS IN SLOT 2
; IT MAY BE A COM OR SERIAL CARD
;
;--------------------------------
RINIT       LDA SLTTYPS+2       ;WHAT CARD IN SLOT 2?
            LDY #020
            BNE GENIT           ;BR ALWAYS TAKEN

;--------------------------------
;
; INIT COM CARD, Y=ONO
;
;--------------------------------
ICOM        LDA #3              ;MASTER INIT
            STA 0C08E,Y         ;TO STATUS
            LDA #21.
            STA 0C08E,Y         ;SET BAUD ETC
CLRIO1      LDX #0              ;CLEAR IORESULT
            RTS                 ;AND RETURN

;--------------------------------
;
; INIT SERIAL CARD, Y=ONO
;
;--------------------------------
ISER        JSR SER1            ;ASSORTED GARBAGE
            JSR 0C800           ;SET UP SLOT DEPENDENTS
CLRIO3      LDX #0              ;CLEAR IORESULT
            RTS                 ;AND RETURN

;--------------------------------
;
; ASSORTED SERIAL CARD SET-UP
;
;--------------------------------
SER1        STY 06F8            ;STORE NO
            TYA
            LSR A
            LSR A
            LSR A
            LSR A
            ORA #0C0
            TAX                 ;MAKE OCN IN X
            LDA #0
            STA TEMP1
            STX TEMP2           ;SET UP INDIRECT ADDRESS
            LDA OCFFF           ;TURN OFF ALL C8 ROMS
            LDA (TEMP1),Y       ;SELECT C8 BANK
            RTS

;--------------------------------
;
; WRITE TO CONSOLE:
; VIDEO SCREEN,COM OR SER CARD IN SLOT 3
;
```

```
;----------------------------------
CWRITE       JSR CONCK          ;CONSOLE CHAR AVAIL?
             BIT CONFLGS        ;IS FLUSH FLAG SET?
             BVS CLRIO          ;YES,DISCARD CHAR & RETURN
             TAX                ;SAVE CHAR IN X
             LDY #030           ;SLOT 3;010
             LDA SLTTYPS+3      ;WHAT KIND OF CARD?
             CMP #3             ;COM CARD?
             BEQ WCOM           ;YES WRITE TO COM CARD SLOT 3
             CMP #4             ;SERIAL CARD?
             BEQ WSER           ;YES,WRITE TO SER CARD SLOT 3
             CMP #6
             BEQ WFIRM
             TXA                ;ELSE RESTORE CHAR & SEND TO SCREEN
VIDOUT       STA TEMP1          ;SAVE CHAR FOR LATER
             JSR INVERT         ;REMOVE CURSOR
             LDY CH
             JSR VOUT2          ;DO THE BUSINESS
             JSR INVERT         ;RESTORE THE CURSOR
CLRIO        LDX #0             ;CLR IORESULT
             RTS                ;RETURN FROM VIDOUT

WFIRM        TXA
             PHA
             LDA #0
             JSR IOWAIT
             JSR SER1
             LDY #0F
             JMP FVEC1

;----------------------------------
;
; WRITE TO SERIAL CARD, Y=ONO,CHAR IN X
;
;----------------------------------
WSER         JSR CONCK          ;CONSOLE CHAR?
             TXA
             PHA                ;SAVE CHAR ON STACK
             JSR SER1           ;ASSORTED GARBAGE
             PLA
             STA 05B8,X         ;SET UP DATA BYTE
             JSR 0C9AA          ;SEND IT (SHOUT)
             LDX #0
             RTS

;----------------------------------
;
; WRITE TO REMOTE:, CHAR IN A
;
;----------------------------------
RWRITE       TAX                ;SAVE CHAR
             LDA SLTTYPS+2      ;WHAT CARD IN SLOT 2?
             LDY #020
             BNE GENW2          ;BR ALWAYS TAKEN

;----------------------------------
;
; WRITE TO PRINTER CARD SLOT1, CHAR IN X
```

```
;
;----------------------------------
WPRN        JSR CONCK           ;CONSOLE CHAR AVAIL?
            LDA OC1C1           ;TEST PRINTER READY
            BMI WPRN            ;LOOP TILL READY
            STX OC090           ;SEND CHAR
CLRIO2      LDX #0
            RTS

;----------------------------------
;
; WRITE TO COM CARD, Y=ONO,CHAR IN X
;
;----------------------------------
WCOM        JSR CONCK           ;CONSOLE CHAR?
            LDA OC08E,Y         ;TEST UART STATUS
            AND #2              ;READY?
            BEQ WCOM            ;NO,WAIT TILL READY
            TXA
            STA OC08F,Y         ;SEND CHAR
            LDX #0
            RTS

;----------------------------------
;
; WRITE TO PRINTER:, CHAR IN A
;
;----------------------------------
PWRITE      TAX                 ;SAVE CHAR IN X
            LDA LFFLAG          ;TEST LINE-FEED FLAG
            BPL LFPASS          ;PASS IF BIT7=0
            CPX #10.            ;IS IT A LINE-FEED?
            BEQ CLRIO           ;YES,IGNORE
LFPASS      LDY #010            ;SLOT 1
            LDA SLTTYPS+1       ;WHAT KIND OF CARD?
GENW        CMP #5              ;PRINTER CARD?
            BEQ WPRN            ;YES WRITE TO PRINTER CARD
GENW2       CMPL#4              ;SERIAL CARD?
            BEQ WSER            ;YES WRITE TO SER CARD
            CMP #3              ;COM CARD?
            BEQ WCOM            ;YES WRITE TO COM CARD
            CMP #6
            BEQ WFIRM
OFFLINE     LDX #9
            RTS

;----------------------------------
;
; READ FROM REMOTE:
;
;----------------------------------
RREAD       LDA SLTTYPS+2       ;WHAT CARD IN SLOT 2?
            LDY #020
GENR        CMP #4              ;SERIAL CARD?
            BEQ RSER            ;GET FROM SER CARD
            CMP #3              ;COM CARD?
            BEQ RCOM            ;GET FROM COM CARD
            CMP #6
```

```
            BEQ RFIRM
            BNE OFFLINE          ;CARD NOT RECOG

;----------------------------------
;
; READ FROM COM CARD, Y=NO
;
;----------------------------------
RCOM        JSR CONCK            ;CHECK FOR CONSOLE CHAR
            LDA 0C08E,Y          ;TEST UART STATUS
            LSR A                ;TEST BIT 0
            BCC RCOM             ; WAIT FOR CHAR
            LDA 0C08F,Y          ;GET CHAR
            LDX #0
            RTS

RFIRM       LDA #1
            JSR IOWAIT
FREAD1      JSR SER1
            PHA
            LDY #0E
            JMP FVEC1

;----------------------------------
;
; READ FROM SERIAL CARD, Y=0NO
;
;----------------------------------
RSER        JSR CONCK            ;CONSOLE CHAR AVAIL?
            JSR SER1             ;ASSORTED GARBAGE
            JSR 0C84D            ;GET A BYTE (SHIFTIN)
            LDA 05B8,X           ;GET BYTE 0678+SLOT
            LDX #0
            RTS

FIRMSTATUS  PHA
            JSR SER1
            LDY #10
            JMP FVEC1

IOWAIT      JSR CONCK
            PHA
            JSR FIRMSTATUS
            PLA
            BCC IOWAIT
            RTS
```

APPLE COMPUTER, INC.
20525 Mariani Avenue
Cupertino, CA 95014

APPLE II PASCAL 1.2
ADDENDUM TO PASCAL TECHNICAL NOTE #11B

(December 1983)

For further information contact:
PCS Developer Technical Support
M/S 22-W, Phone (408) 996-1010

Disclaimer of All Warranties and Liabilities

Apple Computer, Inc. makes no warranties, either express or implied, with respect to this documentation or with respect to the software described in this documentation, its quality, performance, merchantability, or fitness for any particular purpose. Apple Computer, Inc. software is sold or licensed "as is". The entire risk as to its quality and performance is with the vendor. Should the programs prove defective folowing their purchase, the vendor (and not Apple Computer, Inc., its distributor, or its retailer) assumes the entire cost of all necessary servicing, repair, or correction and any incidental or consequential damages. In no event will Apple Computer, Inc. be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the software, even if Apple Computer, Inc. has been advised of the possibility of such damages. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation may not apply to you.

This documentation is copyrighted. All rights are reserved. This document may not, in whole or part, be copied, photocopied, reproduced, translated or reduced to any electronic medium or machine readable form without prior consent, in writing, from Apple Computer, Inc.

Copyright 1983 by Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014
(408) 996-1010

Notice

Apple Computer, Inc. reserves the right to make improvements in the product described in this document at any time and without notice.

APPLE COMPUTER, INC.
20525 Mariani Avenue
Cupertino, CA 95014

APPLE II PASCAL 1.2
ADDENDUM TO PASCAL TECHNICAL NOTE #11

(December 1983)

For further information contact:
PCS Developer Technical Support
M/S 22-W, Phone (408) 996-1010

Contents:

ATTACH Software
- SYSTEM.ATTACH
- ATTACHUD.CODE
- ADMERG.CODE
- CONVAD.CODE
- SHOWAD.CODE
- IM.CODE

Changes to ATTACHUD

Changes to the ATTACH Mechanism
- Transient Initialization

Interrupt Handling
- Handling Interrupts in Version 1.2
- Drivers for Devices with Interrupts
- Changes from Version 1.1
- Example of an Interrupt-Based Device Driver

This document is meant to accompany Pascal Technical Note #11 - Apple Pascal 1.1 BIOS Reconfiguration Using Attach. It describes changes and additions to the Apple // Pascal 1.1 ATTACH facilities.

To use the software described in this addendum, you must have version 1.2 of Apple II Pascal.

ATTACH Software

This section describes the files on the ATTACH TOOLS disk which contains the ATTACH facilities provided for the Apple II Pascal system, version 1.2.

SYSTEM.ATTACH — attaches device drivers at startup time, using the information in ATTACH.DATA, and the driver code in the ATTACH.DRIVERS library. This version of SYSTEM.ATTACH is for use with the 64K and 128K Development Systems only. There is a special version for use with the Runtime Systems which is available on the Runtime System diskettes.

The following programs are provided for the creation and handling of the ATTACH.DATA file.

- ATTACHUD.CODE — creates a version 1.2 ATTACH.DATA file from information supplied by the user
- ADMERG.CODE — merges multiple version 1.2 ATTACH.DATA files into a single ATTACH.DATA file
- CONVAD.CODE — converts an ATTACH.DATA file from version 1.1 to version 1.2
- SHOWAD.CODE — shows the contents of a version 1.2 ATTACH.DATA file

The interfaces to the utilities ADMERG, CONVAD, and SHOWAD are self-explanatory, and we don't describe them in this addendum.

IM.CODE -- contains the interrupt manager (IM) for the 64K Pascal system

Changes to ATTACHUD

If you are familiar with the 1.1 version of ATTACHUD, you will find that the 1.2 version contains some additional prompts. After the question

```
Do you want this driver aligned on
a particular byte boundary? (Y/N)
```

ATTACHUD asks the new question

Do you want this driver to have a
transient initialization section? (Y/N)

If you respond with "Y", ATTACHUD will go on to ask you for the .PROC name of the transient initialization code, and its alignment requirements.

ATTACHUD also asks the new question

Will this driver use interrupts? (Y/N)

If you answer "Y" to this question, ATTACHUD will ensure that a record for the interrupt manager (IM) is present at the end of the ATTACH.DATA file.

Finally, note that unit numbers 13-20 are now available to user-defined devices. These numbers correspond to block-structured devices, and they must be controlled by user-written attach drivers.

Changes to the ATTACH Mechanism

Transient Initialization

As described in Pascal Technical Note #11, a device driver is attached at boot time. If the driver's data record (created by ATTACHUD) specifies that the driver should be initialized at startup time, then its initialization code is executed.

Under version 1.2, there is an additional step. The driver may be accompanied by a "transient initialization" module that is executed only at startup time.

After all the drivers are loaded onto the heap and initialized, each of the transients will be loaded and executed in the same order as their associated driver was loaded. They will overlay each other, going away after completion.

Each of the transients will have passed to it, on the stack, the address of the associated driver. This way communication can be set up between the two. Note that this is the address of the start of the driver, not start - 1.

In order to help data stucturing, the transient code may be loaded on a 0 to 256 byte boundary. Transients must be written in assembler, not use .ABSOLUTE (must be relocatable), and have a single .PROC at the beginning. The transient initialization code must be assembled as a separate module from the device driver itself. Like a device driver, it must be placed in the ATTACH.DRIVERS file using the LIBRARY utility.

Note that the transient initialization code is executed after the device driver's own (callable) initialization code is executed.

This facility was provided for the use of the Pascal ProFile Driver, but it is available to any user-defined device driver.

Interrupt Handling

Version 1.2 of Apple II Pascal supports interrupts from multiple devices.

The first part of this section describes interrupt handling on the Apple II. The second part discusses how to write a device driver that supports interrupts. A sample scheme for such a driver appears at the end of this section.

Important: The interrupt manager (IM) is shipped in the file IM.CODE. For the 64K Pascal systems, the IM driver must be placed in ATTACH.DRIVERS if any devices are to use interrupts. For the 128K Pascal systems, interrupt handling is built in, and the system will ignore the IM driver if it is present in ATTACH.DRIVERS.

The 48K runtime systems cannot use interrupts.

Handling Interrupts in Version 1.2

The main problem in handling interrupts is to save the context of the current program, and then restore that context once the interrupt has been processed. This includes saving the contents of various system registers, and restoring them once the driver returns.

When an interrupt can come from one of several devices, it is also necessary to identify which device, so that the appropriate driver can handle the interrupt.

A driver for a user device that supports interrupts must contain a section of code called the "interrupt service routine." This code will be called by the interrupt manager, as described below.

The interrupt manager (IM) itself is responsible for saving the current context and restoring it later. The interrupt service routines themselves are responsible for determining whether they should handle a given interrupt (just how they do this depends on the particular device; see below).

Interrupt service routines are set up in a linked chain (see item 3 in the following section). If an interrupt service routine recognizes an interrupt, it processes it and then returns to the IM. If the service routine doesn't recognize an interrupt, it transfers control to the next interrupt service routine in the chain. If none of the service routines claims an interrupt, then an error has occurred, and the system is restarted.

Thus, under this scheme, interrupts are handled in the following sequence.

- A device interrupt occurs. This disables interrupts and causes the processor to execute the code that starts at the address stored in the IRQ vector (located at $FFFE-FFFF).

- The IRQ points to the IM, which looks at the processor status on the stack and checks the break bit. If the break bit is set, the IM tranfers control

to the Pascal reset code which restarts the system.

- If the break bit is not set, the IM saves the current context and then transfers control to the first interrupt service routine in the chain.

- If the service routine doesn't recognize the interrupt, it transfers control to the next service routine in the chain. Otherwise, it processes the interrupt and then returns to the IM.

- If the last interrupt service routine in the chain doesn't recognize the interrupt, it transfers control to the reset code for the Pascal system.

- When the IM regains control, it restores the interrupted program's context which re-enables interrupts. Execution proceeds from the point at which it was interrupted.

A spurious interrupt can be generated as the result of a hardware malfunction, or of a BRK instruction in currently executing code. In the case of a hardware malfunction, the interrupt falls through the chain of routines, and control is ultimately passed to the Pascal system reset code. In the case of a BRK instruction, the break bit is set causing the IM to restart the system as decribed above.

To determine whether it should process an interrupt, an interrupt service routine can (in general) check the interrupt flag register for the appropriate card slot.

The location of the interrupt flag register, unfortunately, may vary according to the hardware; it is best if the peripheral card follows the conventions described in the _Apple IIe Design Guidelines_ manual, in the section on "Peripheral Card Firmware."

For 64K Pascal systems, the code for the IM is in the form of an ATTACH driver. However, the IM cannot be called from a user program. (For 128K Pascal systems, interrupt handling is built in, and the IM code is ignored if it is present in ATTACH.DRIVERS).

User-written device drivers that support interrupts must also be ATTACH drivers. The following section discusses how to write such a driver.

Drivers for Devices with Interrupts

The following considerations must be taken into account when you write a driver for a device that generates interrupts.

1. Any volume number appropriate for a user-defined device (128-143) can be used, _except_ for number 128 (decimal), which has been defined as the standard number for the large disk driver. The IM itself is assigned the highest available number. It is recommended that you use numbers in the 130-140 range.

   Note: If you use ADMERG, there is a chance of cancelling another driver that had already been installed with the same number, so it is important to

use SHOWAD to look at the ATTACH.DATA files before you run ADMERG.

2. SYSTEM.ATTACH enables interrupts after the full chain of interrupt service routines has been built and all transient initialization modules have been executed. Device driver code should never enable interrupts.

   In addition, if you wish to execute some code with interrupts disabled, this should not be done with just an SEI instruction. Instead you should use the sequence of PHP, SEI ..code.. PLP. This ensures that the system state is correctly restored when you exit the critical section (after the PLP).

3. Any driver that uses interrupts must initialize itself before the system starts up in order to link its interrupt service code into the chain of service routines. The initialization code should do the following (before exiting) in order to initialize the links:

```
            LDA     0FFFE           ; move IRQ vector into next
            STA     STOREIT         ; driver pointer
            LDA     0FFFF
            STA     STOREIT+1

            LDA     I_ADDRESS       ; move interrupt service routine
            STA     0FFFE           ;  address into the IRQ vector
            LDA     I_ADDRESS+1
            STA     0FFFF

I_ADDRESS   .WORD  I_HANDLER
STOREIT     .WORD  0                ; next driver pointer
```

   where: I_HANDLER is the entry point of the driver's interrupt service routine;

   STOREIT will contain the address of the next interrupt service routine to be called if the current one finds that its device did not generate the interrupt.

   Note: This code must be executed only once and must not be in a transient initialization module. The driver itself may also contain "regular" initialization code to reset the device or its buffer, and so forth.

4. At the start of its interrupt service routine(s), a device driver must first determine whether the driver's device hardware generated the interrupt.

   The details are device-dependent, but in general involve checking a register on the device's controller card (for example, an interrupt flag register on a 6522).

   If the interrupt was generated by the driver's device, the driver should process the interrupt and then return to the IM by an RTI instruction.

   If the interrupt was not generated by the driver's device, the driver should do an indirect jump to the next device driver (the address of the next driver is saved as STOREIT in the sample initialization code under item 3, above). If this device driver is the last in the chain, the jump will be to

the Pascal system reset code.

Note: The jump to Pascal system reset code is accomplished automatically, since the system initializes the IRQ vector to point to the reset code. If the initialization for all interrupt-based device drivers is correct (as shown in item 3), this pointer will be moved to the end of the interrupt service routine chain.

Important: If your device card has no way of signalling that it generated an interrupt, then its service routine must be the last service routine in the chain. It will have to assume that if it is called, it will handle an interrupt. This is not a good approach, since the routine won't be able to detect BRK or hardware failure interrupts.

To ensure that a driver is the last one in the interrupt drivers chain, assign it a unit number lower than all other interrupt driver unit numbers.

5. An interrupt service routine must be an integral part of the driver's code. This ensures that it will be loaded by SYSTEM.ATTACH. If you don't do this, your code is in danger of being released by the system -- a subsequent interrupt may cause unpredictable effects.

6. If you use the 64K Pascal system, the IM driver should be included on the boot diskette inside the ATTACH.DRIVERS file. You may use the standard library program (LIBRARY.CODE) to look into the file and/or transfer the code segment to another file. The code size of IM will be shown as approximately 280 bytes, but much of this size corresponds to relocation code that will not be resident at run time. At run time, the IM occupies approximately 200 bytes.

7. The program ATTACHUD.CODE is used to save information about a driver in ATTACH.DATA. For each driver, ATTACHUD will ask you if the driver uses interrupts. If you answer yes, ATTACHUD ensures that a data record for the IM driver is automatically included in the ATTACH.DATA file. Note that this data record is automatically included in ATTACH.DATA as long as at least one of your drivers uses interrupts.

   On the 64K Pascal system, the IM driver is automatically attached if the IM data record is present in ATTACH.DATA. If the record is not present, the IM driver is not attached. On the 128K Pascal system, the IM data record is ignored if it is present.

8. It is not, repeat not, necessary to save registers in an interrupt service routine. The IM saves them before jumping to the drivers chain, and restores them before resuming normal execution of the interrupted code. You should use the standard 'RTI' instruction at the end of the interrupt service routine: not an 'RTS'. The 'RTI' instruction transfers control back to the IM. (RTI is used because the IM saves additional status information in the processor status byte and then pushes this byte onto the stack.)

9. A change has been made to the 1.2 Pascal system to eliminate a problem associated with abnormal termination of the system with certain interrupting devices. This can occur when a program gets a system error or when a user

interrupts the program from the keyboard (CTRL-@).

When the system terminates abnormally, it executes a UNITCLEAR on all devices (1-20 and 128-143). This is done even when the driver's data record (in ATTACH.DATA) specifies that no initialization is to be done at boot time.

This presents a problem when the UNITCLEAR portion of a driver contains code to initialize the service routine chain (as described above in item 3). Drivers under version 1.2 must have some code to distinguish between the first initialization (which sets up the driver chain) and any subsequent initialization (i.e., a call to UNITCLEAR).

In the driver, these two kinds of initialization may be distinguished by a simple check of a byte of memory to see which type of initialization code needs to be run (if any). This is the scheme used in the example below.

The 1.2 system reinitializes all devices because some drivers may have pointers into the stack/heap space. If this space were released without reinitializing the device drivers, the pointers would now point to invalid code or data. The problem can't be solved by simply disabling further interrupts, since some external devices (e.g., a remote network printer server) may have to be notified of the reset; if interrupts were disabled, information coming back from the remote device could not be handled correctly.

10. Location $7F8 must contain the value $Cn, where n is the slot number of the card, if your card uses the $C800 expansion space. The reason for this is that when you are executing in your $C800 space and an interrupt occurs, the interrupt routine may decide to use its own $C800 space. When the interrupt has completed, the system must know if it needs to reselect the $C800 space for your card. The IM will take the contents of location $7F8 (which can be initialized any time before your driver enters the $C800 space), and use it to reselect your card. If you do not do this, it is very possible that your routines may not work correctly since your $C800 space will not be reselected. The only other way to avoid this is to disable all interrupts while you are in your $C800 space.

    Note: Interrupt service routines must never alter the contents of location $7F8, as this may cause the wrong $C800 space to be reselected after the interrupt has been serviced.

11. Interrupts are disabled when an interrupt occurs, and are re-enabled by the IM after the interrupt has been serviced. Only one interrupt may be handled at a time.

    Devices or drivers must never re-enable interrupts if they have been disabled by the IM.

12. There are additional restrictions on interrupts for applications that execute under the 64K Pascal system and that also use the auxiliary 64K memory on a IIe. Since the IM and all interrupt service routines are resident in the main RAM, if an interrupt occurs while the application is using the auxiliary RAM, the interrupt will not be serviced properly, and

may cause the system to crash.

For this reason, an application should disable interrupts while the auxiliary 64K is in use, or should be able to handle the interrupt management itself.

13. On the Apple IIe, the IM will save the state of the 80STORE and PAGE2 soft switches, and will deselect PAGE2 if 80STORE is selected. The original state of the PAGE2 switch is restored after the interrupt is serviced.

14. In the 128K Pascal system, the IM will save the state of the RAMRD and RAMWRT soft switches and will then select read main RAM and write main RAM. The original state is restored after the interrupt is serviced.

15. If an interrupt service routine uses any zero-page user temporaries ($0-$35), then it must save their contents, and restore them after the interrupt has been serviced.

16. If an application switches in the Monitor ROM, it must disable interrupts prior to doing so.

The following is a brief scenario for installing an interrupt-based device driver in your Pascal system.

1. Write the device driver and assemble it, according to the requirements given above.

2. Execute ATTACHUD to create an attach data file for your driver.

    If you have already defined other device drivers, call the new attach data file INTERRUPT.DATA, for example. Then execute ADMERG to append your driver data file INTERRUPT.DATA to the existing ATTACH.DATA file.

    If you do not have any other device drivers in your system, call the new attach data file ATTACH.DATA.

   Be sure to tell ATTACHUD that your driver uses interrupts.

   Next, execute LIBRARY.CODE to place your driver code in the ATTACH.DRIVERS file. (On the 64K Pascal system, you must include IM.CODE in ATTACH.DRIVERS, if it is not already present.)

   Note: If you change your device driver and reassemble it, you don't always need to run ATTACHUD a second time. Changes to driver code don't affect the data record in ATTACH.DATA unless you have changed something which affects the answer to one of the questions which ATTACHUD asks you. You will still have to use LIBRARY to place the new code in the ATTACH.DRIVERS file.

3. Along with the standard files for a bootable Pascal disk, the following files <u>must</u> be on your new boot diskette: SYSTEM.ATTACH, ATTACH.DRIVERS, and ATTACH.DATA. When you boot from the new diskette, the driver will be loaded, and you can test it and use it.

Remember that you can use SHOWAD to view the contents of ATTACH.DATA, and LIBRARY to view the contents of ATTACH.DRIVERS.

Changes from Version 1.1

Under version 1.1, interrupts were theoretically allowed, since the system disabled interrupts during time-critical operations such as disk accesses. Unfortunately, when a disk access was completed interrupts were never re-enabled, so that interrupts functioned correctly only until a program's first disk access!

Version 1.1 could support only one interrupting device per system.

This is the scheme described in Pascal Technical Note #11.

Version 1.2 of the Apple II Pascal system can support multiple interrupting devices. For 128K systems, this capability is built in. For 64K systems, the interrupt manager (IM) is shipped in the file IM.CODE.

Example of an Interrupt-Based Device Driver

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;               Apple II Pascal 1.2 Sample Interrupt Driver
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;    Copyright 1983 - Apple Computer Inc.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;    This sample driver is a user-defined device driver.  It shows both
;    the overall skeleton of a user driver and more importantly it shows
;    how to write an interrupt-based device driver that uses the
;    interrupt manager (IM).
;
;
;       Macro Subroutines

; Save/restore word off the stack (used to save return addresses)
        .MACRO  POP
        PLA
        STA     %1
        PLA
        STA     %1+1
        .ENDM
;
        .MACRO  PUSH
        LDA     %1+1
        PHA
        LDA     %1
        PHA
        .ENDM
; The ol' switch macro (see SOS Reference Manual for description)
        .MACRO  SWITCH
        .IF     "%1" <> ""
        LDA     %1
        .ENDC
        .IF     "%2" <> ""
        CMP     #%2+1
        BCS     $010
        .ENDC
        ASL     A
        TAY
        LDA     %3+1,Y
        PHA
        LDA     %3,Y
        PHA
        .IF     "%4" <> "*"
        RTS
        .ENDC
$010    .ENDM
; Move first word into the second
```

```
        .MACRO  MOVE
        LDA     %1
        STA     %2
        LDA     %1+1
        STA     %2+1
        .ENDM

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;    Equates
;
;       Zero page (0-$35 is available) is used for return addresses,
;       and global temps.
;

; Zero page temporary locations

CSLIST  .EQU    0                   ; Buffer address
CTRLWORD .EQU   2                   ; storage for ctrl word

IRQ     .EQU    0FFFE               ; IRQ vector location
FLAG6522 .EQU   0C2ED               ; Interrupt Flag Register
                                    ;   for a hypothetical card in Slot 2

;
; Error code equates
;
; Upon completion of the driver, the X register will hold an appropriate
; error code that will be converted into the Pascal reserved variable
; IORESULT.  The Pascal program should check IORESULT after all UNITSTATUS
; calls made to the driver.  Error code numbers 128-255 are to be used by
; your driver.
;

XNOERRS .EQU    0                   ; no errors encountered
XBADCMD .EQU    3                   ; bad command to driver
ERRCODE .EQU    128.                ; user defined error message

        .PROC   SAMPLE

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;    This is the main entry point for the ATTACH driver.  This driver
;    is defined as a 'user device' and therefore will only be used from
;    Pascal using direct I/O (i.e, UNITSTATUS, UNITREAD/WRITE).
;
;    Upon entrance, the X register will contain the type of call requested
;    (UNITREAD,WRITE,CLEAR, etc).  See ATTACH documentation for more
;    details on the stack setup.
;

START
        POP     RETURN              ; save return address
        TXA                         ; get type of call
```

```
        SWITCH  ,4,IDTABLE

BADREQ  LDX     #XBADCMD            ; if you got here, the call is in error!
        BNE     GOBACK              ; always taken
GOBACKOK LDX    #XNOERRS            ; go here if you want to return with no errs
GOBACK  PUSH    RETURN              ; or return with X register holding error code
        RTS                         ; main exit point

IDTABLE .EQU    *
        .WORD   READ-1
        .WORD   WRITE-1
        .WORD   INIT-1
        .WORD   BADREQ-1
        .WORD   U_STATUS-1


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;    INIT does two things: 1) moves the IRQ vectors to the appropriate
;    locations the first time called and 2) every additional call will be
;    meant to issue an appropriate initialization request to the driver
;    (if required).

INIT
        PHP
        SEI                         ; Disable Interrupts
        LDA     TYPE
        BEQ     $001                ; if zero then do init stuff
        .
        .
        Any UNITCLEAR call after the initial one by the system will jump
        to this area
        .
        .
        JMP     $090                ; always taken
$001    INC     TYPE                ; bump type field so next time we dont do it

;
;  This next section moves the IRQ vector into a temporary
;  location.  The MOVE macro is a 16-bit move instruction -- see above
;  macros for an explanation.
;

        MOVE    IRQ,JUMPTO
        MOVE    INTADR,IRQ     ; patch IRQ location and jump vector
        .
        .
        more code for initial initialization call
        .
        .
$090    PLP
        JMP     GOBACKOK
INTADR  .WORD   INTHNDLR            ; Interrupt handler address
JUMPTO  .WORD   0                   ; save area for next interrupt svc routine
```

```
TYPE    .BYTE   0                   ; if 0 then init call else cleanup call
RETURN  .WORD   0                   ; return address for Pascal


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;    READ is called when the program generates a UNITREAD
;
READ
        .
        .
        code for the UNITREAD call
        .
        .
        LDX     #ERRCODE            ; error completion code
        JMP     GOBACK

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;    WRITE is called when the program generates a UNITWRITE
;
WRITE
        .
        .
        code for the UNITWRITE call
        .
        .
        LDX     #ERRCODE            ; error completion code
        JMP     GOBACK

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;    U_STATUS is called when the program executes a UNITSTATUS call to
;    this particular device.
;
;    The order of the stack (4 bytes) is:
;
;    TOS =>  POINTER TO STATUS RECORD
;
;            CONTROL WORD     bits  15..13   12..2         1          0
;                                   user     reserved    status/   direction
;                                  defined   for future  control
;
;                               direction - 0 = status of output channel
;                                           1 = status of input channel
;                             status/ctrl - 0 = status call
;                                           1 = control call
;
;                             Bits 13-15 should have the number of the
;                             control/status request.
;
;
;
```

```
;    From Pascal, the call should be:
;
;              UNITSTATUS(130,BUFFER,OPTION)
;
;    where:
;         130 = Device driver number (currently 130 is used by this driver)
;         BUFFER = PACKED ARRAY [0..??] OF 0..255:
;              This array is as big as needed by the code called.
;         OPTION = PACKED RECORD
;                     DIRECTION : 0..1;
;                     STAT_CTRL : 0..1;
;                     RESERVED : 0..2047;
;                     CODE : 0..7;
;                   END;
U_STATUS
        POP     CSLIST          ; see above
        POP     CTRLWORD        ; ditto
        LDA     #02             ; mask for status/control bit
        BIT     CTRLWORD        ; if bit 2 is set, zero flag will be cleared
        BNE     CONTROL         ; go do a control call
        JMP     STATUS          ; status request

;
;  This is the unitstatus call -- control request section
;
CONTROL
        LDA     CTRLWORD+1      ; get control word
        AND     #0E0            ; is it #0    :  0000 0000
        BEQ     CODE_ZERO       ; yes
        CMP     #20             ; is it #1    :  0010 0000
        BEQ     CODE_ONE        ; yes
        CMP     #40             ; is it #2    :  0100 0000
        BEQ     CODE_TWO
        CMP     #60             ; is it #3    :  0110 0000
        BEQ     CODE_THREE
        LDX     #XBADCODE       ; completion code error
        JMP     GOBACK
CODE_ZERO
        .
        .
        code for control code zero
        .
        .
        LDX     #ERRCODE        ; completion code error
        JMP     GOBACK          ; and report it
CODE_ONE
        .
        .
        code for control code one
        .
        .
        LDX     #ERRCODE        ; completion code error
        JMP     GOBACK          ; and report it
```

```
; repeat for codes 2 and 3
;
CODE_TWO
        .
        .
        LDX     #ERRCODE        ; completion code error
        JMP     GOBACK          ; and report it
CODE_THREE
        .
        .
        LDX     #ERRCODE        ; completion code error
        JMP     GOBACK          ; and report it

;
;  This is the unitstatus call -- status request section
;
STATUS
        LDA     CTRLWORD+1      ; get control word
        AND     #0E0            ; is it #0    :  0000 0000
        BEQ     SCODE_ONE       ; yes
        CMP     #20             ; is it #1    :  0010 0000
        BEQ     SCODE_TWO       ; yes
        LDX     #XBADCODE
        JMP     GOBACK
SCODE_ONE
        .
        .
        status code one
        .
        .
        LDX     #ERRCODE        ; completion code error
        JMP     GOBACK          ; and report it
SCODE_TWO
        .
        .
        status code two
        .
        .
        LDX     #ERRCODE        ; completion code error
        JMP     GOBACK          ; and report it

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;    This is the interrupt handler.  Remember that we don't have to save
;    the context of the system.  The code used to check for an interrupt
;    will have to be changed depending on your hardware.
;

INTHNDLR

;
; first we check to see if we generated the interrupt
;
```

```
        LDA     FLAG6522        ; check for 6522 interrupt flag
        AND     #80             ; mask out bit
        BPL     GOAHEAD         ; if bit was set, we generated it
        JMP     NEXT            ;  otherwise goto the next interrupt handler
;
; Since we generated the interrupt, we service it and then return to
; the interrupt manager with an 'RTI'
;
GOAHEAD
        .
        .
        interrupt handler code for our card
        .
        .
        RTI                     ; Go back to the Interrupt Manager

;
; If we got to NEXT, we must have decided that the interrupt was not
; generated by us.
;
NEXT
        JMP     @JUMPTO
        .END
```

Apple Pascal Object Module Format

Pacal Technical Note # 16

15 October 1981

INTRODUCTION

This document describes the object module format of codefiles currently produced in the Apple][and /// Pascal systems. The only difference between the format of the][and /// codefiles is the information contained in block 0 as noted below. The P-code for both systems is identical.

A CODEFILE ON DISKETTE

Codefiles may be unlinked files created by the compiler or assembler, library files with units which may be used by programs in other codefiles, or linked files composed of segments ready for execution. All codefiles (linked and unlinked) consist of a segment dictionary in block 0 of the file followed by a sequence of one or more code segments up to a total of sixteen segments.

Segments may be linked or unlinked code segments, or data segments for an intrinsic unit. Code segments may have interface text, code blocks, and linker information in that order in blocks on the diskette, though some of these parts may be present only for particular types of code segments. For example, interface text is only present in code segments of units. Data segments only have an entry in the segment dictionary: they do not occupy any blocks on the diskette since they have no code, interface, or linker information associated with them. The only difference between the format of][and /// codefiles is the information in block 0.

Each code segment begins on a boundary between diskette blocks (the 512-byte disk allocation quantum used by the Apple Pascal operating system). Each segment may occupy many blocks (up to a maximum of 32K bytes). A typical codefile is shown in Figure 0.

The following sections describe the parts of a codefile in greater detail. First the segment dictionary is described. Then the parts of a code segment are presented in the order in which they would occur in a file: the interface part, the code part, and finally linker information. The code part description is broken up into sections describing the similarities and differences between code parts for P-code and assembly language modules.

SEGMENT DICTIONARY: BLOCK ZERO OF A CODEFILE

The segment dictionary in block 0 of a codefile contains information regarding name, kind, relative address and length of each code segment. It is

represented by a record presented below in a pseudo-code presentation and illustrated in Figure 1.

The segment dictionary contains an entry for each code or data segment in the file. (The userprogram main segment is assigned segment number 1; the system main segment is assigned segment number 0. Both are placed in slot 0 of their respective codefiles by the compiler. This differs from the statement in the "Pascal Operating System Manual", page 250, which incorrectly states that "the main program is assigned segment #0".)

Each segment dictionary entry includes the segment's size (in bytes). This size is set to zero if there is no segment in the slot. The entry also contains the segment's disk location, which is set to 0 for a data segment of an intrinsic unit. Blocks in a codefile are numbered sequentially from 0, 0 being the segment dictionary. The disk location for non-data segments is given as the block number of the first block containing code for the segment.

```
RECORD {This record is composed of parallel 16 element arrays, one element for
each possible segment slot in the segment dictionary of a codefile.}

       DISKINFO:  ARRAY[0..15] OF
          RECORD
             CODELENG, CODEADDR:  INTEGER
          END;

          {The first array is composed of two-word records made up of two
          integers representing the length of the code part of a segment in
          bytes and the block number of the start of the code part of the
          segment. On the diskette, the CODEADDR field appears before the
          CODELENG in each pair.}

       SEGNAME:  ARRAY[0..15] OF PACKED ARRAY[0..7] OF CHAR;
          {This is a sixteen element array of eight character arrays which
          describe the segments by name. These eight characters are those
          which identify the main program and its segment procedures at compile
          time. Unused segment slots have name fields filled with eight
          ASCII space characters; if the name is less than eight characters it
          is padded on the right by spaces; if the name is longer than eight
          characters, it is truncated. Note that a blank field is allowed for
          an existing code segment. CODELENG=0 should be used to determine an
          empty slot.}

       SEGKIND:  ARRAY [0..15] OF

                 {The next array describes the kind of segment in the
                 particular entry location of the dictionary. The possible
                 values are described below.}

                 (LINKED, {=0. This represents a fully executable segment.
                    Either all external references (regular UNITS or EXTERNALS
                    or .REFs) have been resolved, or none were present.}

                 HOSTSEG, {=1. This represents the outer block of a Pascal
```

program if the program has unresolved external references.}

SEGPROC, {=2. A Pascal segment procedure. This type is not currently used.}

UNITSEG, {=3. A compiled regular (as opposed to intrinsic) unit.}

SEPRTSEG, {=4. A separately compiled (set of) procedures or functions. Assembly language codefiles are always of this type.}

UNLINKED-INTRINS, {=5. An intrinsic unit containing unresolved calls to assembly language procedures or functions.}

LINKED-INTRINS, {=6. An intrinsic unit in its final, ready-to-run state.}

DATASEG); {=7. A specification of the data segment associated with an intrinsic unit telling how many bytes to allocate and which segment to use.}

TEXTADDR: ARRAY[0..15] OF INTEGER; {This array of integers gives the block number of the start of the interface part of each regular or intrinsic unit. The last block of the interface section is inferred from CODEADDR-1. Array elements corresponding to non-unit segments have the value zero. Segments are stored with their interface blocks (if any) first, followed by their code part blocks and finally their linker information blocks (containing symbol table elements for items used but not defined in the segment or for items defined in the segment and externally accessible.) Linker informations records are described in detail below.}

SEGINFO: PACKED ARRAY[0..15] OF
{This array has one word per segment entry.}

PACKED RECORD

SEGNUM: 0..255

{Bits 0 through 7 (the low order bits) of each word specify the segment number for that code. This is the position the code segment will occupy in the system's SEGTABLE at execution time. This table is 32 elements long in the Apple][and 64 elements long in the Apple ///. Thus valid numbers for the first field are 0..31 on the][and 0..63 on the ///.

The run time segment table contains an entry for each segment that is used in executing the program. There are entries for 6 segments that the system uses when executing a user program on the][; on the ///, 8

segments are used by the Pascal operating system. There is an entry for each segment in the segment dictionary of the program's code file. Finally, there is an entry for each code and data segment of each intrinsic unit.

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 number is assigned to a program segment when an entry is created for it in the code file's segment dictionary. The main program has segment number 1. The segments used by the system are 0 and 2..6 on the][and /// and, additionally, 62 and 63 on the ///. Also, segments 59 through 61 are reserved for use by the system. The segment number of an intrinsic unit is determined by the unit's he~~ading when~~ the unit is compiled. (These numbers~~-can be found~~ by examining the segment dictionary of the SYSTEM.LIBRARY file with the LIBMAP or LIBRARY utility programs.) The segment numbers of regular unit segments and of segment pocedures 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.

Since the Pascal system itself uses 6 slots on the][and 8 slots on the /// in the runtime SEGTABLE, this means that a program can have 26 user defined or intrinsic segments (6+26=32) on the][. A codefile is, as we have seen, limited to 16 segments by the number of spaces in the segment dictionary; this is true for both user codefiles and the SYSTEM.LIBRARY codefile. Thus on the][, 16 of the 26 can be in the user's codefile while the excess over 16 could be intrinsics. On the ///, there are 64 possible segments. However, the maximum which can be used is 56: 8 for the system, a maximum of 16 for the user program, up to 16 user program library code or data segments, and up to 16 system library code or data segments. 8+16+16+16=56.

Thus, segment numbers of the program itself, the segments used by the Pascal system, and of any intrinsic units used by the program are fixed before the program is compiled; the segments 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, users need to specify segment numbers only when writing an intrinsic unit. The choice must avoid the fixed numbers 0..6 (and 59 through 63 on the ///) or any other intrinsic unit which may be used in the same program as the unit being written. In particular, the "magic units" PASCALIO and LONGINTIO occupy segments numbers 30 and 31.

Intrinsic unit segment numbers must also avoid conflict with numbers which may be assigned automatically to regular units and segment procedures. However, when unavoidable conflicts arise, the "Next Segment" compiler option described in the "Apple Pascal Language Manual Addendum" may be used to set the segment number to another value.}

MTYPE: 0..15;

{The second byte in the SEGINFO word has in bits 8 through 11 the "machine type" which tells what kind of code is present in the code segment. The machine types are:

0 Unidentified code. Perhaps from a previous compiler.

1 P-code, most significant byte first.

2 P-code, least significant byte first. A stream of packed ASCII characters fills the low byte of a word first, then the high byte. This is the kind of P-code used by Apple.

3 through 9 Assembled machine code, produced from assembly-language text. Machine type 7 identifies machine code for Apple's 6502.}

UNUSED: 0..1;

VERSION: 0..7

{The version number of the system. On the Apple][the current version number is 2; on the Apple /// the current version number is 3.}

END;

INTRINS-SEGS: {ON THE][} SET OF 0..31;
{ON THE ///} SET OF 0..63;

{These words (two on the][, four on the ///) tell the system which intrinsic units are needed in order to execute the codefile. Each intrinsic unit in SYSTEM.LIBRARY (and in the program library on the Apple ///) is identified by a segment number (or two segment numbers if the intrinsic unit has a data segment.) Each of the bits in these words correspond to one of the thirty-two or sixty-four possible intrinsic segment numbers. If the n-th bit is set to 1, this indicates the program needs the intrinsic unit whose segment number in SYSTEM.LIBRARY (or in the program library on the Apple ///) is n.}

INT-NAM-CHECKSUM: {Only on the ///}
PACKED ARRAY[0..63] OF 0..255;

{These fields contain eight-bit checksums of the names of intrinsic units needed to run the codefile. Each entry corresponds to one of the sixty-four possible intrinsic segment numbers on the ///.

The checksum is calculated by shifting the characters of the UNIT name to upper case and adding up the resulting ASCII values of the characters of the UNIT name MOD 256. The name is padded with spaces on the right if it is shorter than eight characters; it is truncated to eight characters if it is longer than eight characters. Padding spaces are included in the checksums. Elements corresponding to unused segment numbers are set to zero.

These words are not used on the][; they must be zeroed.}

{UNUSED JUNK (FILLED WITH ZEROES) FOLLOWED BY}

COMMENT: PACKED ARRAY [0..79] OF CHAR
{the text following a Comment compiler option, starting in byte 432 of the header}

END;

THE INTERFACE PART

Code segments for units may have an INTERFACE part before their associated code blocks. This contains the ASCII text of the INTERFACE declaration in the source code of the UNIT. The construction of an INTERFACE part of a code segment from its source code is shown in Figure 2.

The Pascal compiler emits two block pages (1024 bytes) of text which are identical to the source text blocks except for the first and last pages. The information in the first page is moved up so the first character in the page is the character following "INTERFACE" in the original source. This may leave a considerable amount of unused characters in the first page. Useful information is terminated by a CR and followed by at least one ASCII NULL character (byte value 0). The last page is truncated after the token "IMPLEMENTATION"; it is possible that only one block of this page may be put out if "IMPLEMENTATION" occurs in the first block of the page.

There is some special encoding after the token "IMPLEMENTATION." The immediately following ten characters are composed of ASCII spaces except for an "E" in the ninth position. This is required by the Pascal compiler and librarian program to terminate the interface section. A "P" may occur instead of a space in the second of the ten character positions to signify to the Pascal compiler that the unit requires the PASCALIO unit. The fourth position may be occupied by an "L" if the unit requires the LONGINTIO unit. Failure to include these can cause the system units not to be loaded when needed causing a system crash. Note that these items—IMPLEMENTATION, E, P, and L—are all taken to be tokens by the compiler; thus, the order is significant, the spacing and case is not.

The INTERFACE text is not stripped of excess non-printing characters or

comments and is accessed by the compiler when the UNIT is USEd by another program. Leaving the comments in the INTERFACE part could lead to more complete internal program documentation but may increase size of codefile. This text is not necessary for execution.

The address of the INTERFACE part is given as a block number relative to the start of the code file in the TEXTADDR field described below. This field is zero for segments which are not UNIT code segments or do not have an interface text.

CODE PARTS

As has been mentioned, all non-data segments appear on the diskette as the text of an interface part (if the segment is a regular or intrinsic unit) followed by code blocks followed by linker information (if the segment has undefined elements or has elements which may be linked to other modules.) Data segments for intrinsic units do not occupy any disk blocks.

All code parts have the same general format illustrated in Figure 3. Each code part contains code for that segment's outer block, as well as the code for each of the (non-segment) procedures within that segment. Following code for various procedures associated with the segment is the procedure dictionary at the high address indicated by the CODELENG field of the associated entry in the segment dictionary in block 0 of the codefile. This procedure dictionary grows down; the code starts at the first byte of the block specified in the CODEADDR field of the segment dictionary and grows up.

Each procedure in a code part is assigned a procedure number starting at 1 for the outer block (the main program or segment procedure) and ranging as high as 160. All references to a procedure are made via its number. Translation from a procedure's number to the location of that procedure's code in the code segment is accomplished via the procedure dictionary at the end of the segment. This dictionary is an array indexed by the procedure number. Each array entry is a self-relative pointer to the code for the corresponding procedure. [Since the procedure dictionary starts at the high end of a code segment and works down toward lower addresses, the term "self relative pointer" could be ambiguous: it could be positive or negative depending on interpretation! In all that follows, a self relative pointer is taken to be the absolute distance (in bytes, a positive integer number) between the low order byte of the pointer and the low order byte of the word to which it points.] In other words, you subtract the pointer from its location to find the word pointed to.

Since zero is not a valid procedure number, the zero-th entry of the dictionary is used to store the segment number of the code segment in the low order (even) byte and the number of procedures in that code segment in the high order (odd) byte. The segment number corresponds to the value in the SEGNUM field of the segment dictionary entry.

There are currently two forms of code contained in procedures: P-code and assembly language (or TLA for "The Last Assembler", the familiar name of the assembler currently in use in the Apple Pascal system). Each procedure's code

section consists of two parts: the procedure code itself (in the lower portion of the section growing up toward higher addresses) and a table of attributes of the procedure pointed to by the entry in the procedure dictionary. This table of attributes is loosely known as the Jump Table (JTAB), a term more properly used to refer only to a portion of the table in P-code procedures. The format of the attribute table for a TLA procedure is very different from that for a P-code procedure. These formats are described in the following two sections.

While the compiler and the assembler produce "pure" P-code or TLA code sections, it is possible to produce segments with mixed procedure code type using the Linker. In this case the MTYPE field in the segment dictionary is set to the value for assembly language code, because the code for that segment is now machine specific. The interpreter is able to determine the type of code of a particular procedure via information contained in the procedure's attribute table as is discussed below.

P-CODE PROCEDURE ATTRIBUTE TABLES

The format of a P-code attribute table is illustrated in Figure 4. The contents of the P-code attribute table are:

PROCEDURE NUMBER: Low order, even byte of the word pointed to by the segment dictionary entry. Refers to the number given this procedure in the procedure dictionary of the parent code segment.

LEX LEVEL: High order, odd byte of the same word. Specifies the absolute lexical nesting level for the procedure.

ENTER IC: A self-relative pointer (again, a positive number, pointing back) to the first p-code instruction to be executed for the procedure.

EXIT IC: A self-relative pointer to the beginning of the block of p-code instructions which must be executed to terminate the procedure properly.

PARAMETER SIZE: The number of words of parameters passed to a procedure from its caller.

DATA SIZE: The size of the procedure's activation record in bytes, excluding the Markstack and PARAMETER SIZE. The activation record includes variables and temporary space used by the procedure.

Between these attributes and the procedure code there may be an optional section called the "jump table". Its entries are addresses within the procedure code (as self-relative pointers). During execution, the JTAB system register points to the attributes and jump table section of the currently executing procedure (points to the byte containing the procedure number).

In executing jumps in P-code, a jump opcode has a single byte operand. This is a signed offset: the high order byte is taken to be the sign extension of bit 7. If the offset is non-negative (a short forward jump), it is added to the

interpreter program counter, IPC. (A value of zero for the jump offset makes any jump a two-byte NOP.) If it is negative (a backward or long forward jump), then the operand DIV 2 is used as a word offset into JTAB to find a self-relative pointer, and the instruction program counter is then set to the byte address (JTAB^[operand DIV 2]) - contents of (JTAB[operand DIV 2]).

ASSEMBLY LANGUAGE (TLA) PROCEDURE ATTRIBUTE TABLES

The format of a JTAB for an assembly procedure is very different from that for a P-code procedure. It is illustrated in Figure 5.

The highest word in the JTAB in an assembly procedure always has a zero in its PROCEDURE NUMBER field. In what was the LEX LEVEL field of a P-code procedure JTAB (the high order byte) is either a zero (indicating that BASE RELATIVE relocation is to be relative to the host program activation record) or a non-zero number (indicating the number of the segment relative to which BASE RELATIVE relocation should take place.) In the case of INTRINSIC units without explicitly specified data segments, the number placed in this field is 1. When the interpreter encounters a zero in the procedure number field as it loads the segment, it realizes it must fix up references in the TLA code according to information contained in the rest of the attribute table.

The second highest word of the attribute table is, as before, the ENTER IC: the self-relative pointer to the first instruction to be executed for this procedure. Following this are four relocation tables used by the interpreter at fix-up time.

Working down from the high address start of the JTAB we encounter in order the BASE RELATIVE, SEGMENT RELATIVE, SELF RELATIVE, and INTERPRETER RELATIVE relocation tables. The format of all of these tables is the same: the highest address word of each table specifies the number of entries (possibly zero) which follow (at lower diskette addresses) in the table. Then follows that many single-word entries, which are self relative pointers to locations in the code which must be "fixed up" by the addition of the appropriate relative relocation constant known to the interpreter at load time.

In the case of the BASE RELATIVE relocation table, the value contained in the interpreter's BASE pseudo-register is added if the LEX LEVEL (high order) byte of the procedure's attribute table is zero; if the byte is non-zero, the relocations will be relative to the segment whose segment number is contained in the field. The BASE register is a pointer to the activation record of the most recently invoked base procedure (lexical level 0). Global (lex level 0) variables are accessed by indexing off BASE. The TLA .PUBLIC and .PRIVATE constructs define those entities whose use results in entries into this table.

In the case of the SEGMENT RELATIVE table, the value of the address of the lowest byte in the segment is added. The TLA .REF and .DEF are the relevant constructs.

SELF RELATIVE items have the procedure address (i.e., the address of the lowest byte in the procedure) added.

INTERPRETER RELATIVE items access the Pascal interpreter procedures or variables. They should never be used.

LINKER INFORMATION

Following the code part of a segment there may be Linker information. The starting location of linker information is not included in the segment dictionary as was the case with the starting location of the interface and code parts. It must be inferred. Linker information items may be present for unlinked code segments (i.e., a segment containing unresolved external references) as well as for segments containing items which may be referenced from other segments (e.g., .PROC and .FUNC elements in assembly language programs which may be accessed as EXTERNAL PROCEDURES and FUNCTIONS.) The Linker information begins at the first block boundary following the last block of code for a segment. It is described in detail below. The linker information is a series of records, one for each unit, routine or variable which is referenced but not defined in the source as well as records for items defined to be accessible from other modules. There are records for the following types of items:

```
     litypes =

{0}     ( EOFMARK,  {end-of link-information marker}

          {External reference types: designates fields to be updated by the
          linker}

{1}     UNITREF, {references to invisibly used units— i.e., a reference in
             one unit to another unit. Used in the case of one non-intrinsic
             unit using another non-intrinsic unit.}

{2}     GLOBREF, {references to external global addresses: results from a
             .REF construct in an assembly language program.}

{3}     PUBLREF, {references to a variable in the global data segment of the host
             program: results from a .PUBLIC in assembly language code or use
             of variables declared in the INTERFACE part of regular units. (They
             are stored in another segment in intrinsic units— the data segment
             of the unit.)

{4}     PRIVREF, {references to variables of an assembly language routine or
             regular unit to be stored in the host program's global data segment
             and yet be inaccessible to the host program. Space is allocated by
             the Linker. Generated by .PRIVATE in assembly language. Also,
             generated by use of global variables declared in the IMPLEMENTATION
             part of regular units. (In intrinsic units, these are also stored
             in the data segment of the unit.)

{5}     CONSTREF, {references to a globally declared constant in the host
             program. Generated by .CONST in assembly language.
```

{defining types: gives linker values to fix references}

{6} GLOBDEF, {Global address location. Generated by .DEF (and .PROC and .FUNC) in assembly language}

{7} PUBLDEF, {A variable location in the host program. Generated by VAR declaration in Pascal}

{8} CONSTDEF, {A host program constant definition. Generated by CONSTANT in Pascal.}

{procedure/function information:
Assembler to Pascal and Pascal to Pascal interfaces}

{9} EXTPROC, { References to procedure declared to be external in Pascal: generated by PROCEDURE...EXTERNAL}

{10} EXTFUNC, { References to function declared to be external in Pascal: generated by FUNCTION...EXTERNAL}

{11} SEPPROC, {Separate Procedure definition to be linked into Pascal: generated by .PROC in assembly language.}

{12} SEPFUNC, {Separate Function definition to be linked into Pascal: generated by .FUNC in assembly language.}

{13} SEPPREF, {Not currently used. Was once used for references to procedures in a "separate unit", a concept which has been removed from the current implementation.}

{14} SEPFREF, {Not currently used. Was once used for references to functions in a "separate unit", a concept which has been removed from the current implementation.}

The exact format of data in the linker information block is dependent on the type of entity. They are described by the following record.

```
OPFORMAT = (WORD,BYTE,BIG);
LIENTRY = RECORD

   NAME:  PACKED ARRAY[0..7] OF CHAR; { The name of the symbol.  }

   CASE LITYPE:  LITYPES OF

      GLOBREF,
      PUBLREF,
      PRIVREF,
      CONSTREF,
      UNITREF,

      SEPPREF, {Not currently used}
      SEPFREF:  {Not currently used}

         (FORMAT:  OPFORMAT; {The format of the operand represented by
```

the named (and currently undefined) symbol. May be BIG, BYTE or WORD. (See page 229 of the "Pascal Operating System Manual".)}

NREFS: INTEGER; {The number of references to this symbol in the compiled code segment. There will be this number of pointers after this record into the code segment. These specify the addresses of references to the symbol.}

NWORDS: LCRANGE; {where LCRANGE is 1..MAXLC, currently MAXINT. This field is meaningful only in the case of a PRIVREF type in which case it is the size of the privates in words.}

GLOBDEF:

(HOMEPROC: PROCRANGE; {which procedure the global definition appears in.}

ICOFFSET: ICRANGE); {The byte offset of the occurence in assembly language. IC stands for instruction count.}

PUBLDEF:

(BASEOFFSET: LCRANGE); {compiler assigned word offset into host program data segment.}

CONSTDEF:

(CONSTVAL: INTEGER); {User's defined value}

EXTPROC, EXTFUNC, SEPPROC, SEPFUNC:

(SRCPROC: PROCRANGE; {PROCRANGE = 1..MAXPROC. MAXPROC is currently 160. This field is the procedure number of this procedure definition in its source segment.}

NPARAMS: INTEGER); {Number of parameters expected (really number of words of parameters expected).}

EOFMARK:

(NEXTBASELC: LCRANGE; {Private variable allocation information— amount of space the host used in its data area. Meaningful only for host segments.}

PRIVDATASEG: SEGNUMBER); {Data segment number associated with intrinsic unit code segment. Otherwise not used.}

If the LITYPE is one of the first case variants, then following this portion of the record is a list of pointers into the code segment. Each of these pointers is the absolute byte address within the code segment of the reference to the variable, UNIT or routine named in the LIENTRY. This pointer list is contained in eight-word records, but only the first ((NREF-1) MOD 8)+1 words of the last record are valid.

APPENDIX: SUMMARY OF IMPORTANT RECORD DEFINITIONS

I. SEGMENT DICTIONARY: BLOCK ZERO OF A CODEFILE

```
RECORD

     DISKINFO:  ARRAY[0..15] OF
        RECORD
           CODELENG, CODEADDR:  INTEGER
        END;

     SEGNAME:  ARRAY[0..15] OF PACKED ARRAY[0..7] OF CHAR;

     SEGKIND:  ARRAY [0..15] OF
               (LINKED,
               HOSTSEG,
               SEGPROC,
               UNITSEG,
               SEPRTSEG,
               UNLINKED-INTRINS,
               LINKED-INTRINS,
               DATASEG);

     TEXTADDR:  ARRAY[0..15] OF INTEGER;

     SEGINFO:  PACKED ARRAY[0..15] OF
           PACKED RECORD

           SEGNUM:  0..255
           MTYPE:  0..15;
           UNUSED:  0..1;
           VERSION:  0..7

           END;

     INTRINS-SEGS:  {ON THE ][} SET OF 0..31;
                    {ON THE ///} SET OF 0..63;

     INT-NAM-CHECKSUM:  {Only on the ///}
        PACKED ARRAY[0..63] OF 0..255;

        {These words are not used on the ][; they must be zeroed.}

     {UNUSED JUNK (FILLED WITH ZEROES) FOLLOWED BY}

     COMMENT:  PACKED ARRAY [0..79] OF CHAR
        {the text following a Comment compiler option, starting in byte 432 of
        the header}

     END;
```

II. LINKER INFORMATION

The linker information is a series of records, one for each unit, routine or variable which is referenced but not defined in the source as well as records for items defined to be accessible from other modules. There are records for the following types of items:

```
litypes =

    ( EOFMARK,  {end-of link-information marker}
    UNITREF, {references to invisibly used units.}
    GLOBREF, {references to external global addresses.}
    PUBLREF,  {references to a variable in the global data segment of the
       host program.}
    PRIVREF,  {references to variables of an assembly language routine to
       be stored in the host program's global data segment and yet be
       inaccessible to the host program.}
    CONSTREF, {references to a globally declared constant in the host
       program.}
    GLOBDEF,  {Global address location.}
    PUBLDEF,  {A variable location in the host program.}
    CONSTDEF,  {A host program constant definition.}
    EXTPROC, { References to procedure declared to be external in
       Pascal.}
    EXTFUNC, { References to function declared to be external in
       Pascal.}
    SEPPROC, {Separate Procedure definition to be linked into Pascal.}
    SEPFUNC, {Separate Function definition to be linked into Pascal.}
    SEPPREF, {Not currently used.}
    SEPFREF); {Not currently used.}
```

The exact format of data in the linker information block is dependent on the type of entity. They are described by the following record.

```
  LIENTRY = RECORD

     NAME:  PACKED ARRAY[0..7] OF CHAR; { The name of the symbol.  }

     CASE LITYPE:  LITYPES OF

        GLOBREF,
        PUBLREF,
        PRIVREF,
        CONSTREF,
        UNITREF,

        SEPPREF, {Not currently used}
        SEPFREF:  {Not currently used}

           (FORMAT:  OPFORMAT; {The format of the operand represented by
              the named (and currently undefined) symbol.  May be BIG, BYTE
              or WORD.)}

           NREFS:  INTEGER; {The number of references to this symbol in the
```

compiled code segment. There will be this number of pointers after this record into the code segment. These specify the addresses of references to the symbol.}

NWORDS: LCRANGE; {where LCRANGE is 1..MAXLC, currently MAXINT. This field is meaningful only in the case of a PRIVREF type in which case it is the size of the privates in words.}

GLOBDEF:

(HOMEPROC: PROCRANGE; {which procedure the global definition appears in.}

ICOFFSET: ICRANGE); {The byte offset of the occurence in assembly language. IC stands for instruction count.}

PUBLDEF:

(BASEOFFSET: LCRANGE); {compiler assigned word offset into host program data segment.}

CONSTDEF:

(CONSTVAL: INTEGER); {User's defined value}

EXTPROC, EXTFUNC, SEPPROC {not used}, SEPFUNC {not used}:

(SRCPROC: PROCRANGE; {PROCRANGE = 1..MAXPROC. MAXPROC is currently 160. .This field is the procedure number of this procedure definition in its source segment.}

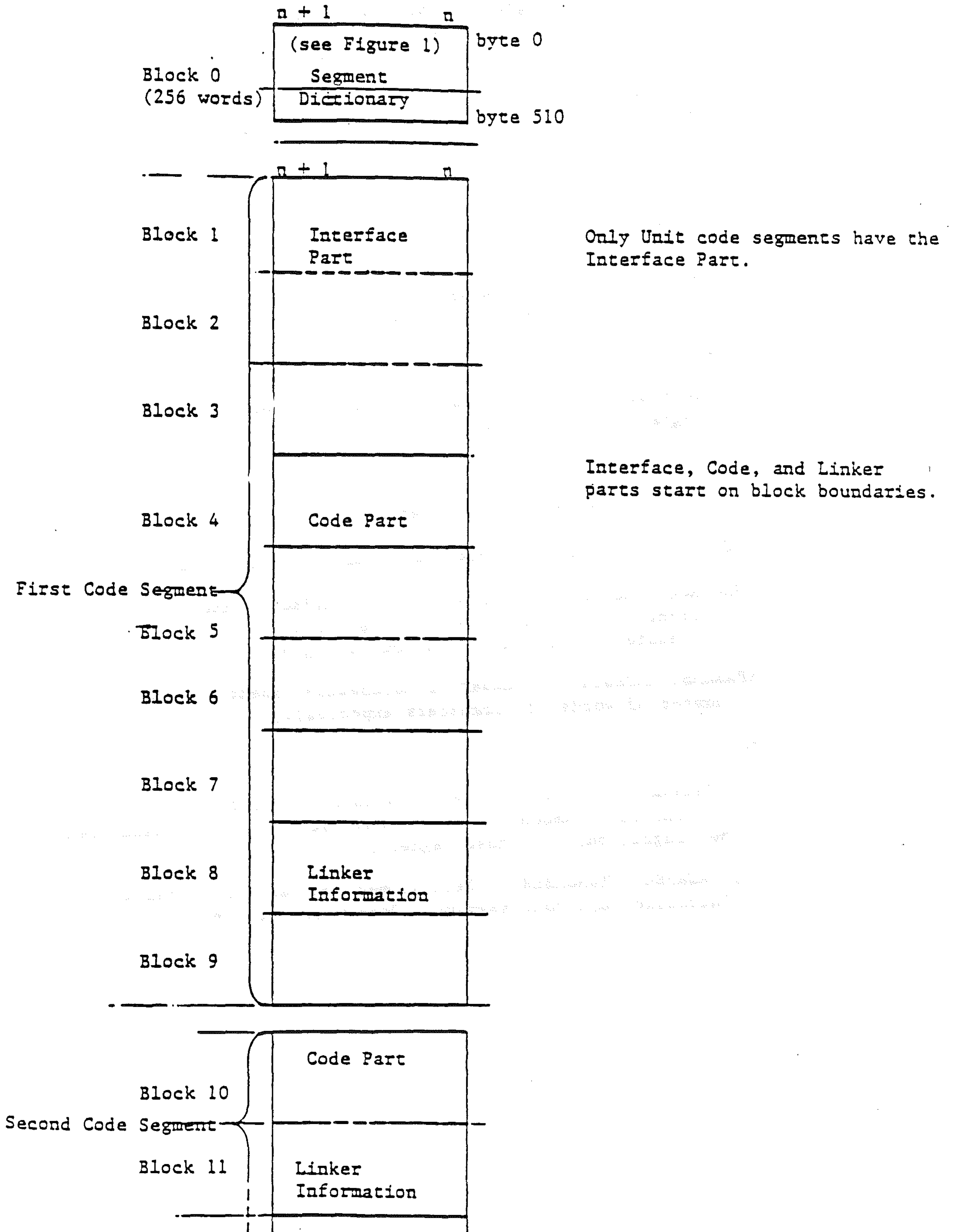
NPARAMS: INTEGER); {Number of parameters expected (really number of words of parameters expected).}

EOFMARK:

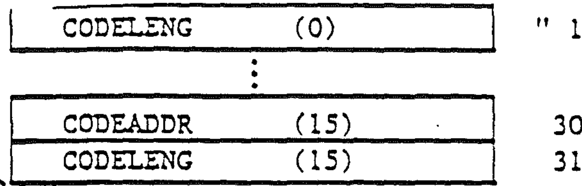
(NEXTBASELC: LCRANGE; {Private variable allocation information— amount of space the host used in its data area. Meaningful only for host segments.}

PRIVDATASEG: SEGNUMBER); {Data segment number associated with intrinsic unit code segment. Otherwise not used.}

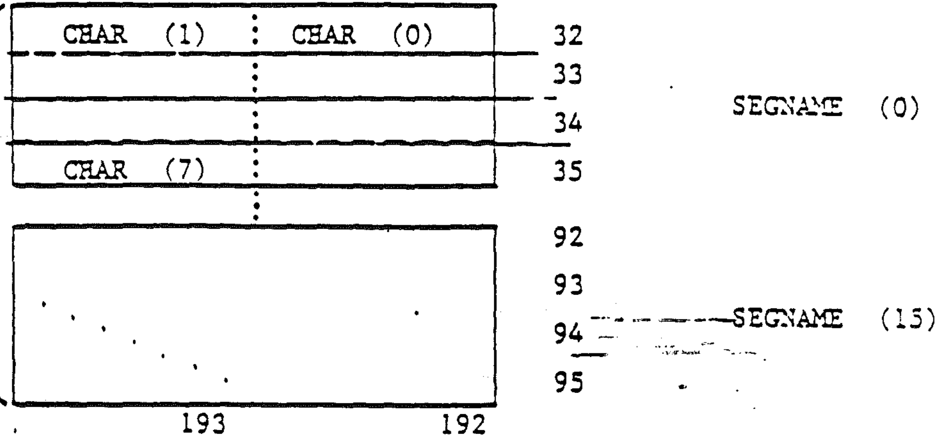
FIGURE 0: THE CODEFILE ON DISKETTE: A TYPICAL CODEFILE



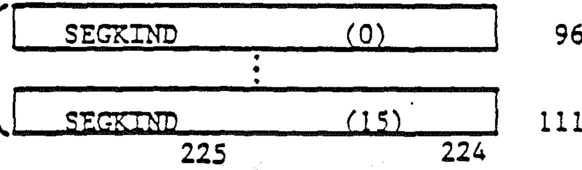
DISK INFO



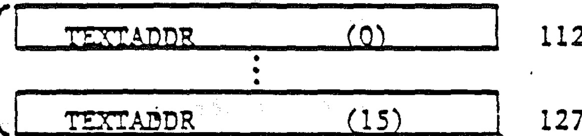
SEGNAME;
Name is truncated if greater than 8 characters; padded with spaces if less than 8 characters



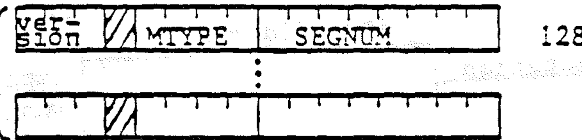
SEG KIND



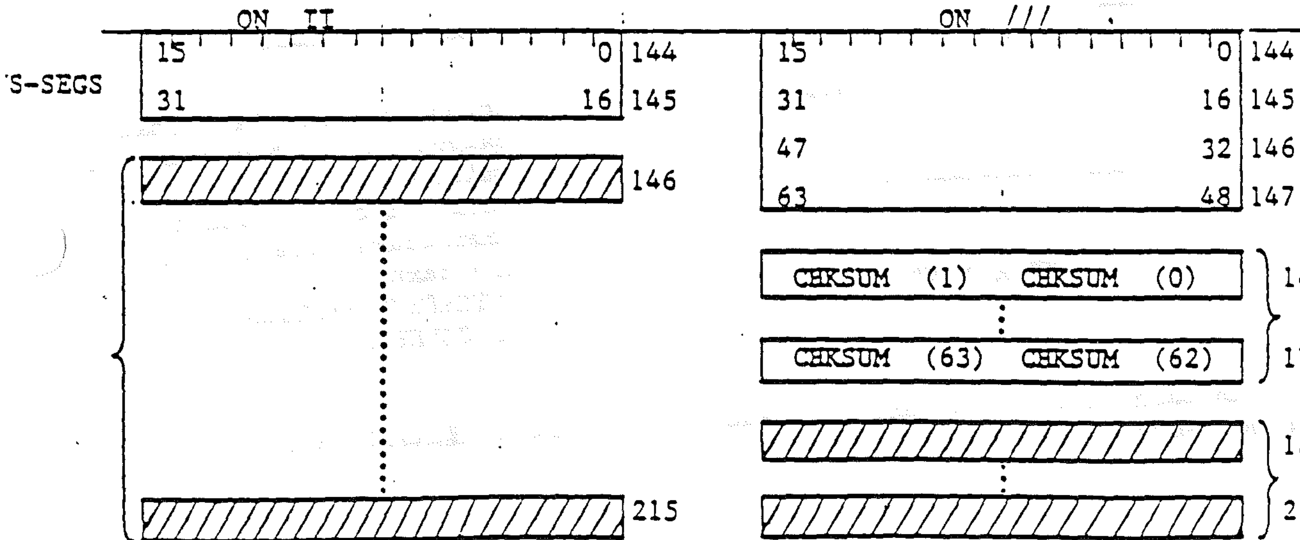
TEXTADDR:
Block Address of interface part text for units.



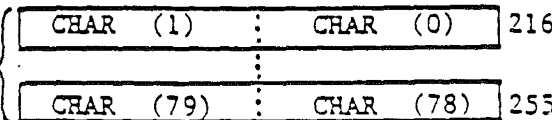
SEGINFO

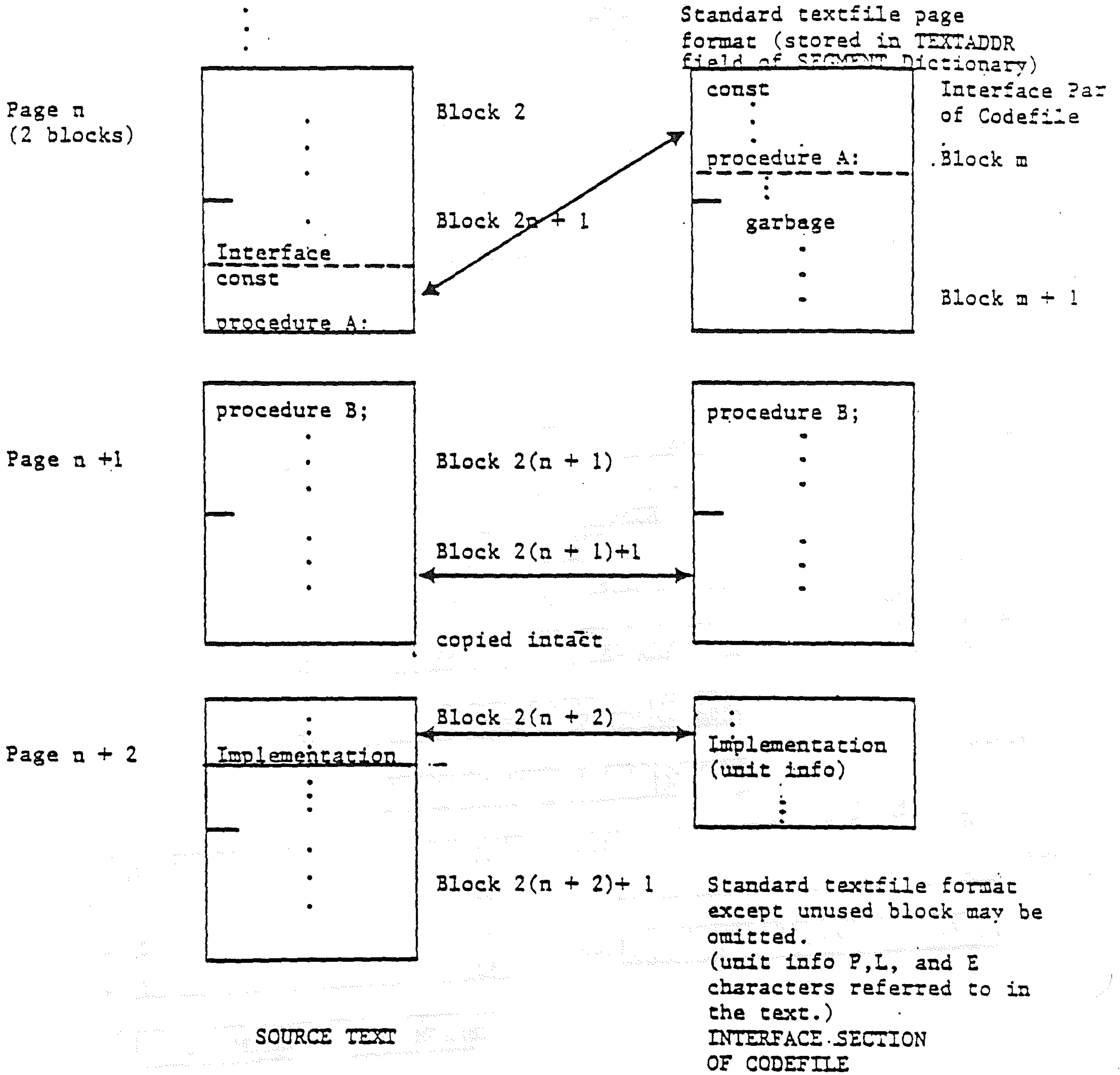


'S-SEGS



COMMENT:
80 characters from comment compiler option





Valid data in each block of a text file end with an ASCII 13, ASCII null character sequence.

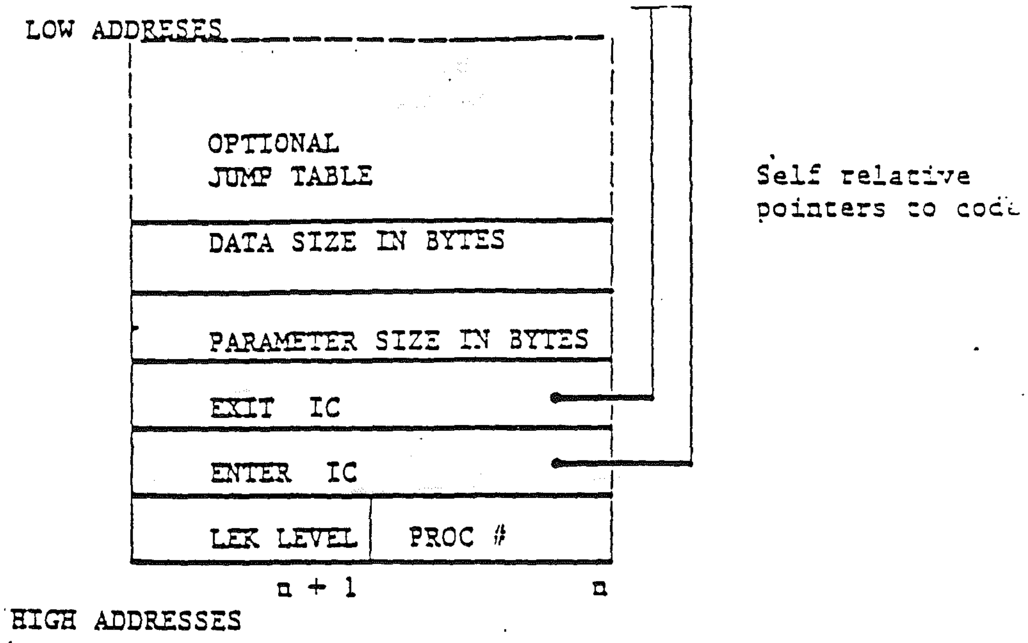


FIGURE 5: TLA ASSEMBLY LANGUAGE PROCEDURE ATTRIBUTE TABLE

LOW ADDRESSES

INTERPRETER
RELATIVE
RELOCATION TABLE

Q WORDS
SELF-RELATIVE
POINTERS

ENTRIES = (Q)

SELF-RELATIVE
RELOCTION
TABLE

P WORDS
SELF-RELATIVE
POINTERS

ENTRIES = (P)

SEGMENT
RELATIVE
RELOCATION TABLE

M WORDS
SELF-RELATIVE
PTRS.

ENTRIES = (M)

BASE RELATIVE
RELOCATION
TABLE

N WORDS
SELF-RELATIVE
PTRS.

ENTRIES = (N)

ENTER IC

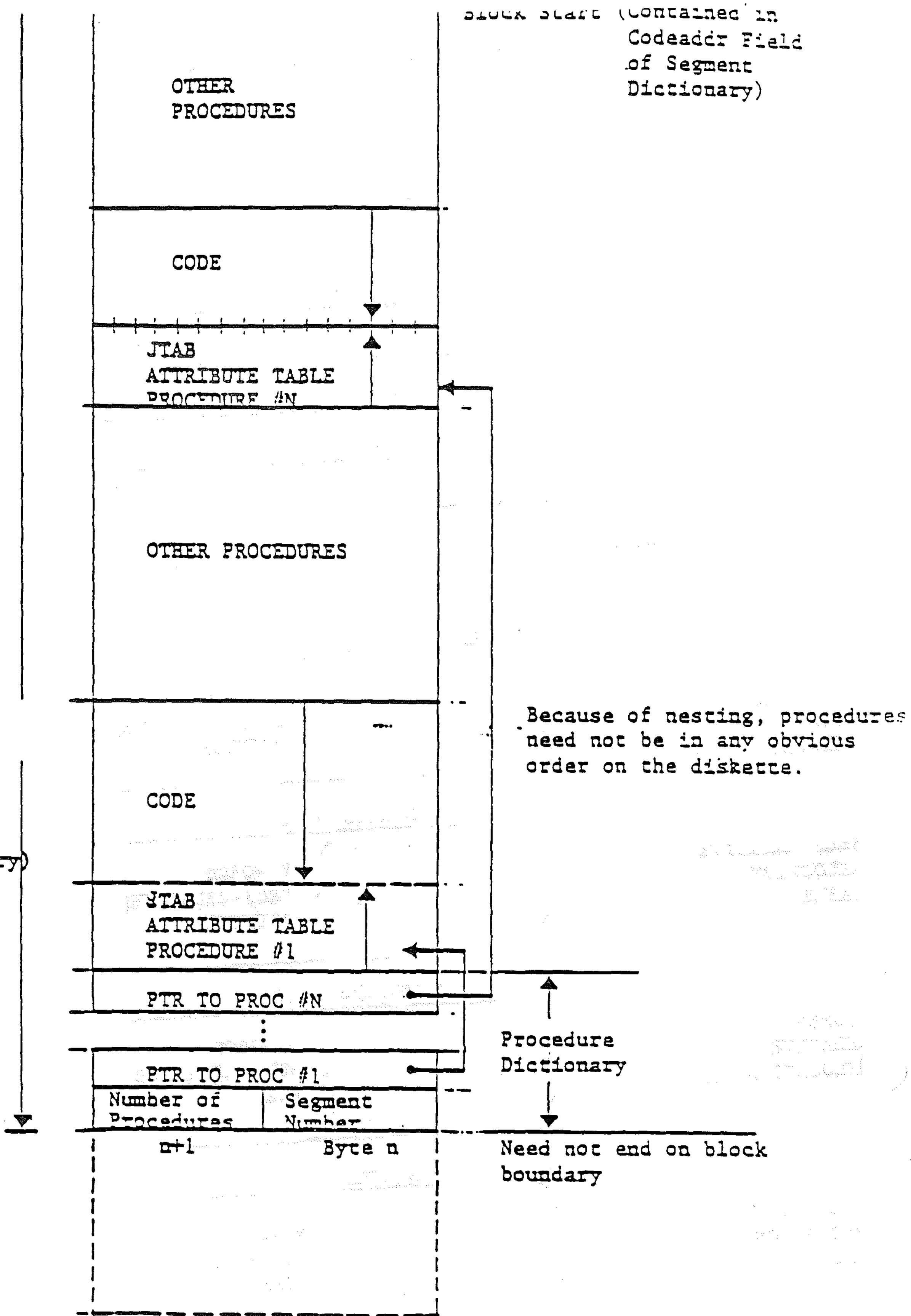
To start of
procedure code

RELOC.SEG.NO.
OR 0

PROC. NUM.
= 0

n + 1
n

HIGH ADDRESSES



Block Start (Contained in Codeaddr Field of Segment Dictionary)
OTHER PROCEDURES
CODE
JTAB ATTRIBUTE TABLE PROCEDURE #N
OTHER PROCEDURES
CODELENG BYTES
ybe several ocks long)
deleng stored segment dictionary)
Because of nesting, procedures need not be in any obvious order on the diskette.
CODE
JTAB ATTRIBUTE TABLE PROCEDURE #1
PTR TO PROC #N
PTR TO PROC #1
Procedure Dictionary
Number of Procedures
Segment Number
n+1
Byte n
Need not end on block boundary
HIGH ADDRESSES
Followed by linker information or by next segment, if any.

APPLE COMPUTER, INC.
20525 Mariani Avenue
Cupertino, CA 95014

PASCAL TECHNICAL NOTE #20
APPLE II PASCAL 1.2
VOLUME MANAGER UNIT TECHNICAL SPECIFICATION
(January 1984)

For further information contact:
PCS Developer Technical Support
M/S 22-W, Phone (408) 996-1010

Disclaimer of All Warranties and Liabilities

Apple Computer, Inc. makes no warranties, either express or implied, with respect to this documentation or with respect to the software described in this documentation, its quality, performance, merchantability, or fitness for any particular purpose. Apple Computer, Inc. software is sold or licensed "as is". The entire risk as to its quality and performance is with the vendor. Should the programs prove defective folowing their purchase, the vendor (and not Apple Computer, Inc., its distributor, or its retailer) assumes the entire cost of all necessary servicing, repair, or correction and any incidental or consequential damages. In no event will Apple Computer, Inc. be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the software, even if Apple Computer, Inc. has been advised of the possibility of such damages. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation may not apply to you.

This documentation is copyrighted. All rights are reserved. This document may not, in whole or part, be copied, photocopied, reproduced, translated or reduced to any electronic medium or machine readable form without prior consent, in writing, from Apple Computer, Inc.

Copyright 1984 by Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014
(408) 996-1010

Notice

Apple Computer, Inc. reserves the right to make improvements in the product described in this document at any time and without notice.

INTRODUCTION

Apple // Pascal release 1.2 only supports the UCSD file format. With the introduction of the Profile for use with the Apple //, the ProDOS operating system has been chosen as the operating system for the support of large mass storage devices. Clearly, the UCSD file format and thus the use of Pascal with a Profile is severely limited (i.e. non-existent) unless there is some means for Pascal to share the resources of the Profile with ProDOS.

The Pascal Profile Manager is a collective term for a set of programs that allow Pascal to share the Profile with ProDOS. These programs supply both user and programatic means to allocate and deallocate Pascal space on the Profile and to assign UCSD file format volumes (known as pseudo-volumes) to the Pascal area of the disk. These pseudo-volumes act analogously to the standard UCSD volumes that currently are found on floppies.

The Volume Manager Unit is a programatic means by which an application program can take advantage of the Profile for the storing of both data and code files. Use of the Volume Manager Unit assumes that the end-user has the PPM program and has already created a Pascal Area on his/her Profile. This unit allows a program to create and manage pseudo-volumes on a Profile. Its functions are:

a. Create a Pascal pseudo-volume.
b. Delete a Pascal pseudo-volume.
c. Assign a pseudo-volume for use.
d. Release a psedo-volume from use.
e. Set or clear write-protection for a pseudo-volume.
f. "Krunch" the Pascal region of the Profile to give space back to ProDOS.
g. Modify the name and/or description field of a pseudo-volume
h. Select the Profile drive to act upon
i. Get the current contents of the Pascal area volume directory
j. Get the current contents of the Profile driver status record
k. Volume Display and Error Reporting

PASCAL USAGE OF THE PROFILE

1. The Pascal Area

The Pascal area of the disk is a contiguous set of blocks that occupies the highest end of the disk, i.e. highest block number down to that block whose number is equal to highest block number minus the total number of blocks that that the Pascal region occupies. This area is not static but expands and contracts as pseudo-volumes are created or deleted and the region is krunched. To insure that Pascal can freely expand, it is a "requirement" that the blocks just below the Pascal region be available and that they be contiguous. Currently a problem may arise if ProDOS has fragmented the disk such that

COPYRIGHT 1984 APPLE COMPUTER, INC

there may be enough logical space for Pascal but not enough contiguous physical space.

The Pascal area is divided into two areas. The first is the Pascal volume directory that specifies the currently allocated Pascal pseudo-volumes in the Pascal area. The second is the pseudo-volumes themselves, each of which having its own volume directory (UCSD format) and its accompanying files.

2. Modifications to the Prodos Directory

The PPM accesses the ProDOS volume directory when it initializes the Profile for use by Pascal. It makes two changes to the directory contents.

The first change is a file entry that specifies the Pascal area on the disk. This file entry is placed in the first available entry slot in the ProDOS volume directory. An error will occur if there is no available slot to put this entry.

Once this slot has been made available, PPM will initialize it with a file entry with the following contents:

```
Stype = 4     this is a ProDOS foreign file structure
name_length = 10
file_name = 'PASCAL.AREA'
file_type = 0EFH  this is a special type to denote the Pascal area
key_pointer = first block used (in this case the second to the
              last block on the disk)
blocks_used = 2
header_pointer = 2
access = 0 (backup bit is not set)
```

All other fields are set to 0. PPM will look for this entry (primarily the name 'PASCAL.AREA') in the ProDOS directory to determine if the disk has been initialized for Pascal use.

The file entry for the Pascal area increments the number of files in the ProDOS directory and the key_pointer for this file now points to TOTAL_BLOCKS - 2. Thus the Pascal area occupies the last two blocks available on the Profile. Blocks_used in the file entry is set to 2. When the Pascal area expands or contracts, the key_pointer and blocks_used values are updated accordingly. With any access to this file entry (i.e. if the Pascal area is expanded or contracted by adding or deleting pseudo-volumes) the backup bit will not be set. However, a ProDOS based Backup program can explicitly backup the Pascal area as a whole. At any time that it cannot expand due to ProDOS using the required blocks, an error is reported. Because ProDOS can fragment its area on the Profile, it is quite possible for Pascal to be unable to expand, though there is logically enough room on the disk to do so. Currently, the only means to correct this is to have the user do the following:

a. backup the Pascal region
b. backup the ProDOS region
c. reformat the disk

d. restore the ProDOS region
e. restore the Pascal region
f. get back to real work

3. Volume Directory Format

The Pascal volume directory contains two separate but contiguous data structures that specify the contents of the Pascal area on the Profile. The volume directory occupies 2 blocks to support 31 pseudo-volumes. It is found at the physical block specified in the ProDOS volume directory as the value of KEY_POINTER, i.e. it occupies the first block in the area pointed to by this value. To access the Pascal area volume directory requires reading the ProDOS volume header via a UNITREAD of block 2, getting the value of KEY_POINTER and using this in a UNITREAD of block number KEY_POINTER. The volume manager maintains a 1K buffer to read in this directory. It is important to define the directory data structures in the volume manager as contiguous to insure that the data read in is interpreted correctly.

The first portion of the volume directory is the actual directory for the pseudo-volumes. It is an array with the following declaration:

```
TYPE  RTYPE = (HEADER, REGULAR)

VAR   VDIR:  ARRAY [0..31] OF
               PACKED RECORD
                 CASE RTYPE OF
                   HEADER: (PSEUDO_DEVICE_LENGTH:INTEGER;
                            CUR_NUM_VOLS:INTEGER;
                            PPM_NAME: STRING[3]);
                   REGULAR: (START:INTEGER;
                             LENGTH:INTEGER;
                             DEFAULT_UNIT:0.255;
                             FILLER:0..127;
                             WP:BOOLEAN;
                             OLDDRIVERADDR:INTEGER
               END;
```

The HEADER specifies information about the Pascal area. It specifies the size in blocks in PSEUDO_DEVICE_LENGTH, the number of currently allocated pseudo-volumes in CUR_NUM_VOLS, and a special validity check value in PPM_NAME, which is a three character string containing the value 'PPM'. The header information is accessed via a reference to VDIR[0]. The REGULAR entry specifies information for each pseudo-volume. START is the starting block address for the pseudo- volume and LENGTH is the length of the pseudo-volume in blocks. DEFAULT_UNIT specifies the default Pascal unit number that this pseudo-volume should be assigned to upon booting the system. This value is set by the volume manager either by the user or an application program and remains valid if it is not released. If the system is shut down, the pseudo-volume will remain assigned and will be active once the system is rebooted. WP is a Boolean that specifies if the pseudo-volume is write- protected. OLDDRIVERADDR holds the address of this unit's (if assigned) previous driver address. It is used when normal floppy unit numbers are assigned to pseudo-volumes so that when released the floppies can be activated again. Each

COPYRIGHT 1984 APPLE COMPUTER, INC

REGULAR entry is accessed via an index (from 1 to 31). This index value is thus associated with a pseudo-volume. All references to pseudo- volumes in the volume manager are made with these indexes.

Immediately following the VDIR array is an array of description fields for each pseudo-volume:

VDESC: ARRAY [0..31] OF STRING[15]

The description field is used to differentiate pseudo-volumes with the same name. It is set when the pseudo-volume is created. This array is accessed with the same index as VDIR.

The volume directory does not maintain the names of the pseudo- volumes. These are found in the directories in each pseudo-volume. When the volume manager is activated, it reads each pseudo-volume directory to construct an array of the pseudo-volume names:

VNAMES: ARRAY [0..31] OF STRING[7]

Each pseudo-volume name is stored here so that the volume manager can use it in its display of pseudo-volumes. The name is set when the pseudo-volume is created and can be changed by the Pascal filer. The names in this array are accessed via the same index as VDIR. This array is set up when the volume manager is initialized and after there is a delete of a pseudo-volume. Creating a pseudo-volume will add to the array at the end.

4. Pascal Pseudo-Volume Format

Each Pascal Pseudo-volume is a standard UCSD format volume. Block 0 and 1 of the pseudo-volume are reserved for bootstrap loaders (which in this case are irrelevent!). The directory for the volume is in blocks 2 through 5 of the pseudo-volume. When a pseudo-volume is created the directory for that pseudo- volume is initialized with the following values:

dfirstblock = 0   first logical block of the volume

dlastblock = 6   first available block after the directory

dvid = name of the volume used in create

deovblk = size of volume specified in create

dnumfiles = 0   no files yet

dloadtime = set to current system date

dlastboot = 0

The Pascal Tech Note #4 describes the format for the UCSD directory.

Files within this subdirectory are allocated via the standard Pascal I/O routines in a contiguous manner.

COPYRIGHT 1984 APPLE COMPUTER, INC

5. Volume Name Format

A valid Pascal UCSD format volume name may be up to seven characters in length and can include any printable ASCII character except ' ', '=', '$', '?', and ','. When ever a user is prompted to enter a volume name, they may enter it in either upper or lower case, however, all lower case letters will be forced to upper case before this volume name is used. For example, if the user enters 'death' as a volume name, it will be uppercased to 'DEATH'.

6. System Limitations

1. Number of Profiles Supported

The Profile driver will currently support only three Profiles, because a Profile interface card can only be plugged into slots 4, 5, or 7 in an Apple //.

The unit numbers used by the Pascal system to refer to the Profiles must be in the range 128 to 143. If they are not, the volume manager will not be able to find them and an error (No Profiles on the system) will result.

2. Number of Pseudo-volumes per Profile Supported

The number of Pascal Pseudo-volumes supported per Profile is limited to 31. This is due to the limitation as to the size of the Pascal area Volume directory. The volume directory is accessed by the Profile driver at boot time in order to assign the default pseudo-volumes. Extending the number of pseudo-volumes supported will require that the driver be changed in order to handle a larger volume directory. Currently the volume directory is 256 bytes.

3. Number of Pseudo-volumes Selected

The Profile driver will allow up to 30 pseudo-volumes to be mounted at any one time. This limit is imposed by the Pascal system as to its number of allowed units. It is reflected in the driver in the data structure STATUS_RECORD. To increase this number requires a change to the Pascal system and to both the Profile Driver and SYSTEM.ATTACH.

4. Pascal Blocked Device Volumes versus User Devices

The Pascal supports the following device numbers as "blocked devices". This implies that they may be accessed like floppies via RESET, REWRITE, READLN, etc.

Blocked Device Unit Numbers

4, 5, 9 - 20

The following unit numbers are for "User devices". They

COPYRIGHT 1984 APPLE COMPUTER, INC

can only be accessed via UNITREAD and UNITWRITE, which implies that Pascal files are not supported for these devices.

User Device Unit Numbers

128 - 143

Thus this system will only support 14 blocked devices on-line at any time. The other 16 volumes are only useful for programs that do their own physical I/O to these volumes. Any floppies attached to the system will use some of the blocked devive unit numbers which leaves fewer of these for pseudo-volumes on the Profile. A user may assign the normal floppie device number to pseudo-volumes, but this will effectively make these floppies inaccessible for use until the pseudo-volumes are released.

5. Volume Name Conflicts

This design allows a user to designate pseudo-volumes with the same name on a single Profile. Many applications may require pseudo-volumes that have the same name, i.e. DATA. In order to support this requirement, we must allow multiple pseudo-volumes with the same name, however, there must be a way to differentiate them both for the user using the Volume Manager Program and for an application program assigning and releasing pseudo-volumes programmatically. To do this, each pseudo-volume entry in the volume directory has an associated description field which is 15 characters in length. This is much the same as extending the volume name by 15 characters.

In order to have pseudo-volumes with the same name on a single Profile, they must have different description fields. For example,

name	description
DATA	QUICKFILE
DATA	PFSREPORT
DATA	MY LIFE STORY

When a pseudo-volume is created this field is specified. When ever a specific pseudo-volume is to be referenced by name, the description field contents are used to differentiate between those with the same name. A user will simply point to the pseudo-volume via cursor motion, using the description field contents displayed as a mnemonic device helping he/she to know which volume is which. A program must pass the expected description contents to the volume manager so it can decide which is which. The rules for finding pseudo-volumes are:

1. If there is only one pseudo-volume with the the name requested then act on that pseudo-volume.

COPYRIGHT 1984 APPLE COMPUTER, INC

2. If there are more than one pseudo-volume with the name requested, then match description fields, return the one matched. If no description content is supplied, return an error.

The volume manager will not allow a user or a program to create pseudo-volumes which cannot be differentiated.

For new applications, it will be important to document how to create and copy their floppy volumes into pseudo-volumes. In this case, the description field will be used by users when hand-assigning Pascal unit numbers prior to execution of the application. Applications that use the volume manager unit can specify this description itself and when assigning unit numbers it can use it to find its own pseudo-volumes.

6. Unit number Conflicts - #4 and #12

Pascal normally allocates its blocked device unit numbers (4, 5, 9 - 12; 13 - 20 are new with Pascal 1.2) to floppies. Unit #4 is normally the floppy drive used to boot the system. It is also by definition, the Pascal system disk which can be referenced via '*'. It is possible to assign this (and any other unit number) to a pseudo-volume. If a user assigns what is normally a floppy drive unit number to a pseudo-volume, they have effectively made that floppy unusable until such time as they release the unit number.

If unit #4 is assigned to a pseudo-volume, the volume manager will then assign unit #12 to the device that was assigned to unit #4. In the usual case, this will be the original boot floppy drive. By doing so, this floppy drive will remain accessible. If unit #4 is assigned and unit #12 is currently not assigned, then unit #12 will be assigned automatically to the device that is normally assigned to be unit #4. Conversely, when unit #4 is released, unit #12 which has been re-assigned will be put back to its original (default) device. If unit #12 is currently assigned (to a pseudo-volume) it will be released and assigned to the normal unit #4 floppy drive. It this case when unit #4 is released, unit #12 will be released from the floppy drive unit, BUT it will NOT be reassigned to its previously assigned pseudo-volume.

This scheme has been adopted because unit #12 is normally assigned to the sixth floppy drive device, which normally does not exist. It will be common practice to assign unit #4 to a pseudo-volume in order to use it as the system "disk" on the Profile. Re-assignment of unit #12 when it has been released from the normal unit #4 floppy drive will only take place if unit #12 was previously assigned to a device, which implies that it has its own driver. Assignment of unit #12 to a pseudo-volume can not be restored.

COPYRIGHT 1984 APPLE COMPUTER, INC

NOTE: If a user assigns unit #4 to a pseudo-volume (which causes unit #12 to be assigned to the boot floppy device) and then assigns unit #12 to a pseudo-volume, this action will make the boot floppy device inaccessible until unit #4 is released.

IMPORTANT NOTE: Assigning a pseudo-volume to unit #4 will immediately release the current unit #4, making it unavailable for use. This may have serious effects. If a program is running that has been invoked from unit #4 (for example, the PPM/volume manager program) and a pseudo-volume is assigned to unit #4, the Pascal operating system will request that the user put in the disk that is in the normal unit #4 device when they exit the program. This is because the system requires the program to be on-line to exit and because the program has been put off-line by the assignment of unit #4. The only recourse is for the user to re-boot the system. Assignment of unit #4 should be done with some thought. Also, if the user intends to place his/her system volume (Pascal development system) in a pseudo-volume and assign this pseudo-volume to unit #4, they must insure that the files for the Pascal operating system occupy the same logical blocks in the pseudo-volume as they occupy on the boot diskette.

If the user assigns a unit number that corresponds to a device that has been configured into the system via ATTACH, that device will become unavailable. Any product that assumes a unit number for a device should warn the user not to assign that unit number to a pseudo-volume when that device must be used.

7. Default Assigning of Pseudo-volumes

The volume directory maintains a mapping of pseudo-volumes to their currently assigned Pascal unit numbers. This assumes that a Pascal area has been initialized, pseudo-volumes have been created in it, and some number of them have been assigned to unit numbers and have not been officially released, i.e. the system has been shutdown without ever releasing these pseudo-volumes. Whenever the system is booted, the Profile driver when activated will read the volume directory from the Profile to determine if and what pseudo-volumes are currently assigned. It will prompt the user

Assign volumes to their default unit number? (Y/N)

and if the user types 'Y' the driver will update its status record to effectively assign these pseudo-volumes to their unit numbers. If the user types 'N' they will not be assigned and will not be accessible.

8. Profile Driver Status Record

The Profile driver maintains a status record that maps Pascal pseudo-volumes to Pascal unit numbers. When a pseudo-volume is assigned, the status record is updated to reflect the assignment. The status record is an array that is mapped into the standard Pascal unit numbers via the mapping

COPYRIGHT 1984 APPLE COMPUTER, INC

PASCAL UNIT NUMBER	INDEX
4	1
5	2
9	3
.	.
.	.
20	14
128	15
.	.
.	.
143	30

The format of the status record is shown below:

```
STATUS_RECORD = ARRAY [1..30] OF
                  PACKED RECORD
                    DRIVE: 0..7;
                    DFMT_DRIVE: 0..7;
                    FILL1: 0..255;
                    WRITE_PROTECT: BOOLEAN;
                    PRESENT: BOOLEAN;
                    START: INTEGER;
                    LGTH: INTEGER;
                  END;
```

When a pseudo-volume is assigned/released, write-protected, and at boot time this status record is updated. Each entry in the status record corresponds to a Pascal Unit number. The field PRESENT, if a 1, connotes that this unit number is assigned. The field DRIVE specifies the Profile drive on which the pseudo-volume resides, START gives the physical block number of the starting block of the pseudo-volume, and LGTH is the length of the pseudo-volume in blocks. WRITE_PROTECT, if a 1, implies that this pseudo-volume is write-protected. DFMT_DRIVE is used to assign the last used drive when the volume manager program/unit is restarted. When the system is booted the default mount drive (DFMT_DRIVE) is set to 0. If the next drive command or SELECT_DRIVE procedure is called, this value is changed to reflect the new drive and stored in the Profile driver. When the volume manager is exited and then at some point re-invoked, it will read this value from the driver and use it as the current drive. If the system is shutdown, this value will revert to 0. This data structure is not intended to be accessed by any program other than the volume manager and the Profile driver itself.

9. Use of the Profile Driver

Both the PPM and the volume manager assume that there is a Profile driver attached and that the name of this driver is 'PROFILE'. At initialization time for both these programs, if no Profile driver is found (identified by its name 'PROFILE') then an error message is issued:

ERROR: There is no Profile driver available for this Pascal system

and the program will terminate.

COPYRIGHT 1984 APPLE COMPUTER, INC

The Profile Driver is supplied as the file ATTACH.DRIVERS and its associated data file is ATTACH.DATA. This driver is configured to be unit #128.

THE VOLUME DISPLAY

The volume display occupies the major portion of the screen and is used to display the pseudo-volumes available for use on the currently selected Profile drive. This display has two uses:

1. Display the pseudo-volumes available

2. Serve as the means to select a pseudo-volume upon which to apply one of the actions in the volume manager command line.

The format for the the volume display is shown below with example pseudo-volumes:

```
Profile drive:  0
WP Name    Description     Unit         WP Name    Description     Unit
*  DATA    QUICKFILE       #9           *  ACCOUNT PFSREPORT       #134
   DATA    DBMSTUFF                     *  DATA    PFSFILE
   LETTERS                 #13             FUN     SOME GAMES
   PASSYS  PASCALSYS       #4              MOREFUN NOT A GAME
   PASDEVO SOME TOOLS      #5
   BOB     OUR SAVIOR
   YHVH1   STARK FIST
   AP      APPLEACCOUNT
   GL      APPLEACCOUNT
   AR      APPLEACCOUNT
   <none>  PFSDATAVOL
   TOOLS   MORE TOOLS      #15
   TEXT    DOCUMENTS       #16
   YETI
   PICTURE TURTLEGRAPHICS  #19
*  RESUME  FUTURES
```

COPYRIGHT 1984 APPLE COMPUTER, INC

This format will allow up to 31 pseudo-volumes to be displayed at one time. If there is less than 17 pseudo-volumes to be displayed, the right hand column header is suppressed.

The first line shows which Profile drive is active by giving the drive number (in this case 0). As other drives are selected, this number will change.

The fields in the display are described below:

WP - this is the write-protect attribute for the pseudo-volume. If it is write-protected, a '*' will be displayed in this column.

NAME - this is the name of the pseudo-volume. It can be up to 7 characters in length. Multiple pseudo-volumes can have the same name if and only if their description fields are different. It is possible for a pseudo-volume to not have a name, i.e. some applications use the entire volume for data wiping out the directory. If no name is found the string "<none>" is displayed.

DESCRIPTION - this is the description field for the pseudo-volume that helps to both differentiate it from others with the same name and also serve as a reminder to the user what the contents of that pseudo-volume are.

UNIT - if the pseudo-volume is currently mounted then its Pascal unit number will be displayed, else this field will be blank.

SELECTING A VOLUME

When an action that affects an individual pseudo-volume is selected from the prompt line, the characters '->' will appear next to the first pseudo-volume displayed. By using the up or down arrow keys (as defined by the Pascal system and machine in use) the user can move the pointer from one pseudo-volume to another. UP will move the cursor up on the screen and DOWN will move it down. The pseudo-volumes are 'numbered' from top to bottom with the first column 'numbered' from 1 to 16 and the second column 'numbered' from 17 to 32. UP moves down the 'numbers' and DOWN moves up the numbers!! If more than 32 pseudo-volumes are allowed in the display then multiple screen pages are used to display the pseudo- volumes. Movement between screen pages is done using the UP and DOWN arrow keys and a to be determined modifier key.

For a standard Apple // system, CTRL-O is defined be UP and CTRL-L is defined to be DOWN. This is the convention followed by Pascal on the Apple //. For the //e the up-arrow and down-arrow keys are respectively UP and DOWN.

COPYRIGHT 1984 APPLE COMPUTER, INC

Once the pointer has been moved to the desired pseudo- volume, typing a RIGHT ARROW will select that pseudo-volume for the action specified in responce to the prompt line. When a pseudo- volume is selected, it will be highlighted. The volume manager may ask for further prompting/information once RIGHT ARROW has been typed. When prompted, typing ESCAPE will cancel both pseudo-volume selection and the action selected from the prompt line.

The right-arrow key on both the Apple // and the //e corresponds to RIGHT ARROW.

THE VOLUME MANAGER UNIT SPECIFICATION

1. Introduction

The Volume Manager Unit (VOLUME_MANAGER) is a programatic interface, to allow developers to write programs that can manage Pascal pseudo-volumes on a Profile. It supplies the following generic capabilities through lower level procedure calls:

a. Create a Pascal pseudo-volume

b. Delete a Pascal pseudo-volume (this is not a recommended practice for application programs to do.)

c. Assign a Pascal Unit number to a pseudo-volume to make it available for use

d. Release a Pascal Unit number from a pseudo-volume

e. Set the write-protection attribute for a pseudo-volume

f. Krunch the Pascal region to give space on the Profile back to ProDOS (this is also not a recommended practice for applications to perform. This ability will be built in to the Delete call as well as being a stand alone procedure.)

g. Modify the name and/or description field of a pseudo-volume.

h. Select the Current Profile drive on which to perform the above actions (an application program would not normally have to do this except to search for a pseudo-volume that it needs to assign.)

i. Get the contents of the Pascal area volume directory. This is for information purposes only. A program cannot change its contents.

j. Get the contents of the status record in the Profile driver. Again this is for information purposes only.

COPYRIGHT 1984 APPLE COMPUTER, INC

k. Utilize the volume display (section 4.4.2 and 4.4.3) to display and select pseudo-volumes.

l. Error reporting.

The user has the option to do their own screen management for input and/or error reporting.

An application program cannot initialize a Pascal region on a Profile. This must be done via the Pascal Profile Manager by the end-user. Applications that require this action will need to document this requirement so that the user of the application can correctly set up the Profile for use.

The volume manager unit is a REGULAR unit and must be linked to a host program.

2. Volume Manager Unit Interface

1. Constants

MAX_VOLS

This is the maximum number of pseudo-volumes that can be allocated in the Pascal area on a Profile. This number is 31.

MAX_DRIVE

This is the highest drive number for use in referencing the Profile drives. This number is currently 7, but only drives 0, 1, 2 are supported.

VDIR_SIZE

This is the size of the volume directory in blocks. For 31 pseudo-volumes its value is 2.

MAXDUNIT

This constant represents the highest unit number for blocked devices which is 20.

2. Types

UNIT_RANGE

This is the range of unit numbers supported by the Pascal system. The range is 0 to 255.

RTYPE

This is used to differentiate the two types of record fields in the volume directory. The two types are HEADER which refers to the header information

COPYRIGHT 1984 APPLE COMPUTER, INC

in the volume directory and REGULAR which refers to the entry used for each pseudo-volume in the directory.

DRIVE_RANGE

This is the range of values for drive numbers used to reference Profile drives. The range is 0 to MAX_DRIVE.

STAT_REC

This is the declaration for the data structure STATUS_RECORD that is found in the Profile driver. It maintains information about the currently assigned Pascal unit numbers. Its format is:

```
  STAT_REC = ARRAY [1 .. 30] OF
             PACKED RECORD
               DRIVE: 0 .. 7;
               DFMT_DRIVE: 0 .. 7;
               FILL1: 0 .. 255;
               WRITE_PROTECT: BOOLEAN;
               PRESENT: BOOLEAN;
               START: INTEGER;
               LGTH: INTEGER;
             END;
```

This structure is described above.

VDIR_STRUCT

This is the format for the volume directory. Its structure is:

```
VDIR_STRUCT = ARRAY [0 .. MAX_VOLS] OF
               PACKED RECORD
                 CASE RTYPE OF
                   HEADER: (PSEUDO_DEVICE_LENGTH:INTEGER;
                            CUR_NUM_VOLS:INTEGER;
                            PPM_NAME:STRING[3]);
                   REGULAR: (START:INTEGER;
                             LENGTH:INTEGER;
                             DEFAULT_UNIT:UNIT_RANGE;
                             FILLER:0 .. 127;
                             WP:BOOLEAN;
                             OLDDRIVERADDR:INTEGER)
               END;
```

This data structure is fully described above.

DESC_ARRAY

This array holds the description fields for each pseudo-volume. It is important that any program that gets the volume directory contents must also declare

this data structure contiguous to the volume directory data structure. Its format is:

DESC_ARRAY: ARRAY [0 .. MAX_VOLS] OF STRING[15]

N_ARRAY

This array will hold the names of the pseudo-volumes. It also must be declared if the application program intends to get the volume directory contents. It does not have to be declared in any special place however. Its format is:

N_ARRAY: ARRAY [0 .. MAX_VOLS] OF STRING[7]

STRING7

This is a string of length 7. It should be used to declare any variable that is to hold a pseudo-volume name.

STRING15

This is a string of length 15. It should be used to declare any variable that us to hold a description field.

BLOCK_TYPE

This is a 512 element array of bytes that is used to hold blocks of data read in from a disk. It is primarily used for low-level routines and is not necessary for application programs.

3. Variables

VALID_DRIVE

This is a set that holds the valid drive numbers for all the available Profile drives. Its format is

VALID_DRIVE: SET OF DRIVE_RANGE

This variable is initialized when the volume manager is activated. If a drive number is in VALID_DRIVES it does not imply that this drive has a Pascal area. It only implies that this drive is active and that it has a ProDOS directory. An application program should use PASCAL_DRIVES to determine if this drive has a valid Pascal area.

PASCAL_DRIVES

This is the set that specifies all the available

COPYRIGHT 1984 APPLE COMPUTER, INC

Profiles that have Pascal areas. All of the volume manager unit functions can only be applied to Profiles that are specified in this set. Any application must check the drive number against this set prior to making any calls to the volume manager unit. Since SELECT_DRIVE must be called prior to making any other calls, it will check the drive number against this set and return an error if it is not in the set. A call to INIT_VM will put together both VALID_DRIVE and PASCAL_DRIVES. The format for this set is

PASCAL_DRIVES: SET OF DRIVE_RANGE

MY_UNIT

This is the unit number by which the Pascal system refers to the Profile driver. It is an integer.

ERR_LINE

This variable holds the line number on which errors are reported. Its value defaults to 3. An application program can change this value. It is only used when ERR_FMT (see below) is TRUE.

DISPLAY_ERR

This boolean variable is used to control whether or not the volume manager willl report errors to the screen. If TRUE, then errors will be reported, else they will not be reported.

ERR_FMT

If this boolean variable is TRUE then errors will be reported on ERR_LINE, else they will be reported on the current line of the display.

VM_ERROR

This integer variable will contain an error code if an error has occurred on a call to the volume manager. If it is 0, then no error has occurred.

VM_IO_ERROR

This integer variable will contain the value of IORESULT after any call to the volume manager. If it is 0 then no error has occurred.

CUR_DRIVE

This is the current drive number for the currently

COPYRIGHT 1984 APPLE COMPUTER, INC

accessible Profile unto which volume manager actions can occur.

CUR_INDEX

This is the index of the currently selected pseudo-volume on the current drive. It is only set via the volume selection routine SEL_VOLUME.

VDIR_BYTES

This is the size of the volume directory plus the description array in bytes. It is used in reading and writing the contents from and to the Profile. It is initialized by the volume manager unit.

VDIR

This is the current copy of the volume directory of the currently selected drive. It is initialized by SELECT_DRIVE.

VDESC

This is the current copy of the array of descriptions that corresponds to the pseudo-volumes of the currently selected drive. It is initialized by SELECT_DRIVE.

VNAMES

This is the current array of volume names for the pseudo-volumes of the currently selected drive. It is initialized by SELECT_DRIVE.

STATUS_REC

This is the current copy of the status record from the Profile driver. It is initialized by INIT_VM.

4. Procedures and Functions

CREATE_VOLUME

Call format:

INDEX := CREATE_VOLUME(NAME, DESC, SIZE)

where NAME is a 7 byte string that will be the name of the volume, DESC is a 15 byte string that denotes the description field (this may be null), and SIZE which is an integer that denotes the number of blocks

COPYRIGHT 1984 APPLE COMPUTER, INC

this pseudo-volume is to occupy. INDEX is a user-supplied integer to hold the index value that is returned.

CREATE_VOLUME will create a pseudo-volume on the currently selected Profile drive. It will be assigned a name, its description field will be specified, and it will be SIZE blocks in length. This function will then return an index value that must be used in any other call to act on this pseudo-volume. It is up to the calling program to save this index value. (It can be found however through a VOLUME_INDEX call described below.) If an error occurs, INDEX will be set to 0. Use of this function will change the index values that correspond to the pseudo-volumes on the Profile.

Errors reported:

a. Not enough room - there is not enough room in the Pascal region to allocate a pseudo-volume of this size or the Pascal region cannot expand into the ProDOS area. A Krunch may alleviate this problem.

b. Directory full - there is no more room in the volume directory to allocate this pseudo-volume.

c. Name conflict - a pseudo-volume with this name already exists and the description field does not differentiate them. This can be solved either by specifying the description field or changing it.

d. Illegal volume name

e. Volume size must be greater than 6 blocks.

DELETE_VOLUME

Call format:

DELETE_VOLUME(INDEX, KRUNCH_FLAG)

where INDEX is an index into the volume directory that specifies which volume to act upon and KRUNCH_FLAG is a Boolean.

DELETE_VOLUME will delete the pseudo-volume specified by INDEX, which corresponds to a pseudo-volume (either through a create or VOLUME_INDEX call) only if it contains to

COPYRIGHT 1984 APPLE COMPUTER, INC

files (if so an error occurs). If KRUNCH_FLAG is set to TRUE, the volume manager will then krunch the Pascal region, else it will not. This procedure follows the name matching convention specified above. Use of this procedure will cause a change in the indexes used to specify pseudo-volumes. If this procedure is used, an application program should update its own copy of the indexes prior to making any calls that use an index.

Errors reported:

a. No such volume - a volume with the INDEX passed was not found.

b. Write-protect error - if the pseudo-volume is write-protected it cannot be deleted

c. Volume has files cannot delete.

ASSIGN_VOLUME

Call format:

ASSIGN_VOLUME(INDEX, UNIT_NUMBER)

where INDEX is an integer and UNIT_NUMBER is an integer in the range 4, 5, 9 - 20, 128 - 143.

This procedure will assign the Pascal unit number (UNIT_NUMBER) to the pseudo-volume specified by INDEX. The unit number must be in the correct range. This action will make the pseudo-volume accessible through the normal Pascal I/O routines. If this unit number is already assigned, the current devive (or volume) will be released from this unit number and the new one will be assigned.

Errors reported:

a. No such volume - a volume with the index passed was not found.

b. Illegal Unit Number - the unit number passed to this procedure was out of range.

c. Cannot assign Profile driver unit number.

RELEASE_VOLUME

Call format:

COPYRIGHT 1984 APPLE COMPUTER, INC

RELEASE_VOLUME(UNIT_NUMBER)

where UNIT_NUMBER is an integer in the range
4, 5, 9 - 20, 128 - 143.

This procedure will release the pseudo-volume assigned to the Pascal unit number (UNIT_NUMBER). Doing so will make this pseudo-volume inaccessible to Pascal I/O calls.

Errors reported:

a. Not assigned - this unit is currently not assigned.

b. Illegal Unit Number - the unit number passed is not in the legal range.

WP_VOLUME

Call format:

WP_VOLUME(INDEX, WP_FLAG)

where INDEX is an integer and WP_FLAG is a Boolean.

WP_VOLUME will set or unset the write-protect attribute of the volume specified by INDEX. If WP_FLAG is TRUE then it will be write-protected else it will be unwrite-protected.

Errors reported:

a. No such volume - there is no volume specified by this index

KRUNCH_AREA

Call format:

KRUNCH_AREA

This procedure will krunch the Pascal region of the currently active Profile.

SELECT_DRIVE

Call format:

SELECT_DRIVE(DRIVE_NUMBER)

where DRIVE_NUMBER is an integer in the range
0 to 7.

COPYRIGHT 1984 APPLE COMPUTER, INC

SELECT_DRIVE will select a Profile for the Volume Manager to act upon. The available set of Profile drives is given in the set PASCAL_DRIVES found in the global variables. All volume manager calls are specific to a single Profile. To switch Profiles requires this call.

Errors reported:

a. Drive not active - this drive is not available for use.

b. Illegal drive number - the drive number passed is out of range.

c. No Pascal area on drive.

MODIFY_VOLUME

Call format:

MODIFY_VOLUME(INDEX, NAME, DESCRIPTION)

where NAME is a 7 character string and DESCRIPTION is a 15 character string, INDEX is an integer

This procedure will modify the name and/or the description field of a pseudo-volume specified by INDEX. Either string passed may be null. This will leave the current contents unchanged. Errors that can occur are:

a. No such volume - there is no such volume specified by this index

b. Illegal volume name

c. Write protect error

d. Name conflict

VOLUME_INDEX

Call format:

INDEX := VOLUME_INDEX(NAME, DESCRIPTION)

where NAME is 7 byte string, DESCRIPTION is a 15 byte string, and INDEX is an integer.

VOLUME_INDEX will look up a volume in the volume directory and return its index, which

COPYRIGHT 1984 APPLE COMPUTER, INC

is then used to perform any volume manager action on that volume. This routine will follow the volume name matching conventions specified above. This call will usually proceed any other volume manager call. Use of these indexes can be made easier if the calling program maintains a mapping between pseudo-volume names and their indicies once this call has been made. After the deletion of pseudo-volume, however, the application cannot assume that the indexes will remain the same.

Errors reported:

a. No such volume - a volume with this name cannot be found.

GET_VIDR

Call format:

GET_VDIR(VOL_DIRECTORY, NAME_ARRAY, DRIVE_NUMBER)

where VOL_DIRECTORY is of type VDIR_STRUCT (defined in Volume manager interface section) and NAME_ARRAY is of type N_ARRAY (also defined in the interface section. DRIVE_NUMBER is an integer in the range 0 to 7.

This procedure will return the contents of the volume directory on the Profile drive designated by DRIVE_NUMBER. The contents are returned in the user supplied data structure VOL_DIRECTORY which is declared to be of type VDIR_STRUCT. The names of the pseudo-volumes are returned in NAME_ARRAY.

It is important to declare in the application program the following data structures in this order and contiguous:

```
VOL_DIRECTORY: VDIR_STRUCT;
DESCRIPTIONS: DESC_ARRAY;
```

because this call will fill both these data structures.

Errors reported:

a. Illegal drive number - must be in the range 0 to 7

b. Invalid drive - this drive is not available

c. No Pascal area on this drive - this Profile does not contain a Pascal area

COPYRIGHT 1984 APPLE COMPUTER, INC

GET_STATREC

Call format:

GET_STATREC(STATUS_RECORD)

where STATUS_RECORD is of type STAT_REC (defined in the interface section of the unit.)

This procedure will return the contents of the status record foound in the Profile driver. This contains information about the currently assigned Pascal unit numbers.

Errors reported:

a. No Profile driver - there is no Profile driver attached

INIT_VM

Call format:

INIT_VM

This procedure will initialize the volume manager unit. It sets various global variables, identifies the Profile driver and its unit number, and sets the value for CUR_DRIVE. It DOES NOT initialize the volume directory or status record data structures. The caller must immediately call SELECT_DRIVE with an appropriate drive number to initialize these data structures prior to making any other calls to the volume manager unit. If the volume manager unit is configured such that it is swapped in and out of memory (NOLOAD option) then this procedure must be called whenever the volume manager unit is swapped back in followed by a call to SELECT_DRIVE. This procedure sets up the sets VALID_DRIVE and PASCAL_DRIVES.

Errors reported:

a. No profile driver attached - this is essentially a fatal error since no actions can occur without a profile.

WP_DISPLAY

Call format:

WP_DISPLAY(INDEX, WP)

COPYRIGHT 1984 APPLE COMPUTER, INC

where INDEX is an integer and WP is a Boolean.

An application may have the volume manager unit display the volume selection screen (shown above) This procedure will update the write-protect field in the display that corresponds to the pseudo-volume specified by INDEX. If WP is true a '*' will be placed in the column or if it is false a ' ' will be placed there.

Errors reported:

a. No such volume - this index value does not correspond to an existing pseudo-volume.

NAME_DISPLAY

Call format:

NAME_DISPLAY(INDEX, NAME)

where INDEX is an integer and NAME is a seven character string.

An application may have the volume manager unit display the volume selection screen (shown above) This procedure will update the name field for the pseudo-volume specified by INDEX with the name passed in NAME.

Errors reported:

a. No such volume - this index value does not correspond to an existing pseudo-volume.

DESC_DISPLAY

Call format:

DESC_DISPLAY(INDEX, DESC)

where INDEX is an integer and DESC is a 15 character string.

An application may have the volume manager unit display the volume selection screen (shown above) This procedure will update the description field for the pseudo-volume specified by INDEX with the string passed in DESC.

Errors reported:

a. No such volume - this index value does not correspond to an existing pseudo-volume.

COPYRIGHT 1984 APPLE COMPUTER, INC

UNIT_DISPLAY

Call format:

UNIT_DISPLAY(INDEX, UNIT_NUM)

where INDEX and UNIT_NUM are integers.

An application may have the volume manager unit display the volume selection screen (shown above) This procedure will update the unit number display for the pseudo-volume specified by INDEX. If UNIT_NUM is a valid UCSD unit number it will update the display to show the number, else it will set the unit number display to blanks (meaning that this pseudo-volume is not assigned.) When a pseudo-volume is released, the display can be updated by calling this procedure with UNIT_NUM equal to 0.

Errors reported:

a. No such volume - this index value does not correspond to an existing pseudo-volume.

VOL_DISPLAY

Call format:

VOL_DISPLAY(INDEX)

where INDEX is an integer.

An application may have the volume manager unit display the volume selection screen (shown above) This procedure will update all the information (write_protect, name, description, and unit number) for the pseudo-volume specified by INDEX.

Errors reported:

a. No such volume - this index value does not correspond to an existing pseudo-volume.

TITLE_DISPLAY

Call format:

TITLE_DISPLAY

An application may have the volume manager unit display the volume selection screen (shown above)

COPYRIGHT 1984 APPLE COMPUTER, INC

This procedure displays the column headings for the volume display.

SCREEN_DISPLAY

Call format:

SCREEN_DISPLAY

An application may have the volume manager unit display the volume selection screen (shown above) This procedure will put the complete volume display on the screen for the currently selected Profile. It requires that SELECT_DRIVE has been called. After any create or delete of a pseudo-volume, this procedure should be called to update the complete volume display.

SEL_VOLUME

Call format:

INDEX := SEL_VOLUME

where INDEX is an integer.

An application may have the volume manager unit display the volume selection screen (shown above) If the volume display is used, this routine can be called to have a user select a pseudo-volume from the display as described in the section above. The pseudo-volume selected is specified by the value returned in INDEX. If INDEX is 0 this specifies that the user has aborted the selection process and that NO pseudo-volume has been selected.

REPORT_ERROR

Call format:

REPORT_ERROR

An application may choose to have the volume manager unit report any errors that may have occurred to the screen. If DISPLAY_ERR is true, this procedure will report an error message to the screen. If ERR_FMT is true, the error messages will be displayed on line 3 else they will be displayed at the current position of the cursor. The error displayed will be based on the value of VM_ERROR or VM_IO_ERROR with VM_ERROR having the highest precedence. If both these values are 0 (no error) then no error message will be displayed. After

a volume manager routine has been called, an application program can then call REPORT_ERROR to report any errors that may have occurred.

S_CLEARSCREEN

Call format:

S_CLEARSCREEN

This procedure will clear the screen. It is supplied as a low-level screen management procedure.

S_CLEARLINE

Call format:

S_CLEARLINE

This procedure will clear the current line (i.e. the line in which the cursor currently lies.) It assumes that the cursor is in column 0.

USING THE VOLUME MANAGER UNIT

1. Introduction

This section reviews in detail the way an application writer will use the Volume Manager Unit. Details for the procedure and function calls are given above. The actions that can be performed with this unit are shown below:

a. Creating A Pascal Pseudo-volume
b. Deleting A Pascal Pseudo-volume
c. Assigning A Pascal Pseudo-volume
d. Releasing A Pascal Pseudo-volume
e. Setting the Write-protection of a Pascal Pseudo-volume
f. Krunching the Pascal region of the Profile
g. Modify the name/description field of a pseudo-volume
h. Selecting the Profile Drive to Use
i. Getting the Index for a Pseudo-volume
j. Getting the Pascal area Volume Directory
k. Getting the Profile Driver Status Record
l. Screen management routines
m. Error reporting

Each of these actions is performed on the current drive selected, thus it is important for the user to know which drive they are performing these actions.

All existing pseudo-volumes are referenced via their volume directory index. This value can be obtained either when a program calls the volume manager to create a pseudo-volume or through a function call to the volume

COPYRIGHT 1984 APPLE COMPUTER, INC

manager given a pseudo- volume's name and description field. Also, a function is supplied that will allow an application program to use the human interface found in the volume manager program.

2. Data Structures

The data structures supplied in the interface section can be divided into 3 areas:

a. Profile information

b. Pseudo-volume directory information

c. Control of display and error reporting

2.1 Profile Information

The volume manager unit maintains a certain data structures that describe the state of the Profile driver. These are:

VALID_DRIVES - the set of all available Profile drive numbers (does not imply that these drives have Pascal areas)

PASCAL_DRIVES - the set of all Profiles with Pascal areas

CUR_DRIVE - the currently selected Profile drive number

STATUS_REC - the Profile driver status record which maps pseudo-volumes to Pascal unit numbers making them available for use

2.2 Pseudo-volume Directory Information

Once a Profile drive has been selected by a call to SELECT_DRIVE, the volume manager unit will maintain directory information for the pseudo-volumes on that Profile. This information is kept in the following data structures:

VDIR - this is the actual volume directory for the Pascal area on this Profile

VDESC - this is the array which holds the description fields for the pseudo-volumes

VNAMES - this is the array which holds the volume names for the pseudo-volumes

The volume manager unit will update both these data structures and their counterparts on the drive itself after any change

COPYRIGHT 1984 APPLE COMPUTER, INC

is made by a call to the volume manager.

2.3 Control of Display and Error Reporting

Use of the volume manager unit's display routines is based on the setting of some control flags:

DISPLAY_ERR - if TRUE the volume manager unit will report errors to the screen on the line specified by ERR_LINE (normally set to 3)

ERR_LINE - the line on which errors are reported

ERR_FMT - if TRUE report errors on ERR_LINE else report them at the current location of the cursor

When errors occur in the volume manager unit, two variables are set to reflect the error condition:

VM_ERROR - this holds an integer that denotes the error that has occurred

VM_IO_ERROR - if an I/O error occurs then this variable will have the value of IORESULT

3. Creating A Pascal Pseudo-volume

To create a Pascal pseudo-volume requires a call of the form:

```
INDEX := CREATE_VOLUME(NAME, DESC, SIZE)
```

This will create a pseudo-volume on the currently selected Profile with the name NAME, its description field will be set to the string passed in DESC, and it will be SIZE blocks in length. The index returned should be stored in the calling program, for it must be used for all other calls that will assign, delete, etc. this pseudo-volume. The index can also be obtained by a call to VOLUME_INDEX using the same name and description field. This call can return 3 possible errors, either to the calling program or by reporting them to the screen (if so desired.)

4. Deleting A Pascal Pseudo-volume

This is not a recommended practice for application programs to do. The end-user should only delete pseudo- volumes via the volume manager program (from the PPM). If an application needs to delete a pseudo-volume, it is done through the call

```
DELETE_VOLUME(INDEX, KRUNCH_FLAG)
```

The index corresponds to a pseudo-volume that is obtained either through a CREATE_VOLUME or VOLUME_INDEX call. After a pseudo-volume has been deleted, the Pascal region can be krunched if the KRUNCH_FLAG is set to TRUE. This call will return an error if there is no volume that corresponds to that index

COPYRIGHT 1984 APPLE COMPUTER, INC

or if the volume is write-protected.

5. Assigning A Pascal Pseudo-volume

For a program to use a pseudo-volume as a Pascal volume, the pseudo-volume must be assigned to a Pascal unit number. To do so requires a call of the form

    ASSIGN_VOLUME(INDEX, UNIT_NUMBER)

The index value specifies the pseudo-volume to assign with the Pascal unit number passed via UNIT_NUMBER. An error will occur if there is no corresponding pseudo-volume or if the UNIT_NUMBER value is not in the correct range of Pascal unit numbers (4, 5, 9 - 20, 128 - 143).

6. Releasing A Pascal Pseudo-volume

To release a pseudo-volume from its assigned Pascal unit number, requires a call of the form:

    RELEASE_VOLUME(UNIT_NUMBER)

where UNIT_NUMBER corresponds to the Pascal unit number that has been assigned. It is recommended that any application that assigns unit numbers will also release them before completion of execution. This will free the user from having to hand-release these pseudo-volumes before executing another program. This call can return two errors, one of which if the unit is not currently assigned or if the unit number is not in the correct range.

7. Setting the Write-protection of a Pascal Pseudo-volume

To set or clear the write-protect attribute of pseudo- volume, make the call

    WP_VOLUME(INDEX, WP_FLAG)

INDEX selects the volume and if WP_FLAG is true it will be write-protected, else the write-protect attribute will be cleared. An error will occur if the is no volume that corresponds to the index passed.

8. Krunching the Pascal Region of the Profile

This is not a recommended practice for application programs. The only time it may be necessary is if when a create of volume is attempted and there is not enough room for the volume a call to KRUNCH_AREA may free up enough space for the volume. The call is simply

    KRUNCH_AREA

9. Modify the name/description field of a Pseudo-volume

An application can change the name and/or the description field of a pseudo-volume. This is not a recommended practice. Calling MODIFY_VOLUME(INDEX, NAME, DESCRIPTION) will change the specified values.

COPYRIGHT 1984 APPLE COMPUTER, INC

Either the NAME or DESCRIPTION parameter may be null, to not change the field.

10. Selecting the Profile Drive to Use

All volume manager actions are performed on the currently selected Profile drive. Each Profile drive is assigned a drive number (in the range 0 to 7). The default Profile is drive 0. To select a Profile, the application program should check the set PASCAL_DRIVES in the volume manager interface to determine which drives are active. PASCAL_DRIVES is set up when the volume manager is initialized. Any currently active Profile drives will be placed in it. If a user turns off a Profile after PASCAL_DRIVES is set, then any action to that Profile will result in an I/O error. For example,

```
IF 0 IN PASCAL_DRIVES THEN SELECT_DRIVE(0)
```

SELECT_DRIVE will return an error if the drive is not active, i.e. if it is not in PASCAL_DRIVES or if an illegal drive number (out of range) is passed.

11. Getting the Index of a Pseudo-volume

In order to act upon a pseudo-volume, you require the index that corresponds to that pseudo-volume. To get the index requires a call

```
INDEX := VOLUME_INDEX(NAME, DESCRIPTION)
```

This function will return the index that corresponds to the pseudo-volume whose name and description field match the values passed. If an error occurs it will return a value of 0 to INDEX. The rules for matching are:

a. if there is only one pseudo-volume with the name NAME then return its index

b. if there are more than one pseudo-volume with the same name, then match description fields. If there is no match then return an error. If there is a clear match then return the index.

c. if no name is matched then return an error.

12. Getting the Pascal Area Volume Directory

Normally, an application program will not have to know about the contents of the Pascal area volume directory. In such cases as it may, this procedure is supplied to allow a program to inspect the contents (but it may not change them.) The program needs to declare the following data structures in its global data section in the following order and format:

```
VAR

    VOLUME_DIRECTORY: VDIR_STRUCT;
    DESCRIPTIONS: DESC_ARRAY;
    NAME_ARRAY: N_ARRAY;
```

Calling GET_VDIR will transfer the information into these data structures. Care must be made that the programmmer does not put any other data structures amidst these for they will be wiped out! Use of this procedure will not set CUR_DRIVE to this drive_number.

13. Getting the Profile Driver Status Record

Using GET_STATREC is also not intended for the usual use of the volume manager unit. Again, this only supplies information and the user cannot change the contents. The program must declare the data structure STATUS_RECORD shown below in its global data area:

```
VAR

    STATUS_RECORD: STAT_REC;
```

14. Error Handling

After any call to the volume manager unit, there is a possibility that an error occurred. This is registered in the VMERROR variable found in the volume manager interface. After any call, this variable should be checked to see if an error has occurred. Any I/O errors are noted in the variable VM_IO_ERROR. It should also be checked. The error values are shown below for VMERROR:

0 - No error

1 - No such pseudo-volume

2 - Not enough room to allocate pseudo-volume

3 - Volume directory full

4 - Name conflict

5 - Illegal unit number

6 - Pseudo-volume not assigned

7 - Profile Drive not active

8 - Illegal drive number

9 - Illegal volume name

10 - Write Protect error

11 - No Pascal Area on this Profile

12 - No Profile driver attached

13 - Volume size must be greater than 6 blocks

14 - ProDOS directory is full

15 - Pseudo-volume contains files cannot delete

16 - Cannot assign unit number used for Profile driver

17 - The ProDOS directory has a ProDOS file called PASCAL.AREA

VM_IO_ERROR will contain the standard IORESULT value for any I/O errors that may have occurred. Use of these two variables parallels the use of IORESULT in Pascal programs. After a call has been made to the volume manager, the application should check VM_ERROR and VM_IO_ERROR, to determine the success of the call.

The application program has the choice whether or not it wishes to report any errors that may occur while using the volume manager unit. Also, it can allow the volume manager unit to report the errors. Two variables found in the interface control error reporting. They are:

DISPLAY_ERROR - if TRUE then the volume manager will report errors to the console, else no error messages will be displayed

ERR_FMT - if TRUE and if DISPLAY_ERROR is TRUE then all error messages will be displayed on lina ERR_LINE which is set to 3, by default, of the console, else if ERR_FMT is FALSE and DISPLAY_ERROR is TRUE then error messages will be displayed on the current line of the console, i.e. at the current cursor position

ERR_LINE - this variable specifies on which line to report errors. It is set to line 3 by default. An application program can change this value to suit its needs. It is only used if ERR_FMT is set TRUE.

The volume manager supplies an error reporting procedure REPORT_ERROR, that will print an error message based on the current values of VMERROR or VM_IO_ERROR. An application program can call this procedure to report any errors. This procedure will report errors given the settings of the above flags.

15. Managing the Screen Display

For the most part, the application program is expected to manage its own screen display a propos to its purposes. The volume manager unit supplies the routines necessary to use the volume display shown in the section above. After an an application has performed a SELECT_DRIVE it can display the available pseudo-volumes on that drive by calling SCREEN_DISPLAY. Various fields within that display can be updated after any volume manager unit call following the protocols given below:

After the creation of a pseudo-volume:

call SCREEN_DISPLAY

After the deletion of a pseudo-volume:

call SCREEN_DISPLAY

After assigning a pseudo-volume:

call SCREEN_DISPLAY

After releasing a pseudo-volume:

call UNIT_DISPLAY with the index of the pseudo-volume that has been released with a unit number of 0

After clearing or setting of write_protection:

call WP_DISPLAY with the index of the pseudo-volume and a boolean where TRUE means write-protection has been set and FALSE means write-protection has been cleared

After krunching:

no update to the screen is necessary

After modifying the name or description field:

call either/both NAME_DISPLAY and/or DESC_DISPLAY with the index of the pseudo-volume and the new value for that field

After selecting a Profile drive:

call SCREEN_DISPLAY

To have the user select a pseudo-volume:

once SCREEN_DISPLAY has been called, call SEL_VOLUME to have the user select a pseudo-volume, this call will return its index

An application can use the volume manager unit's error reporting mechanism if it so chooses. If it chooses to do it itself, the variables VM_ERROR and VM_IO_ERROR are available to the application program to use to determine what if any error has occured and to report it in its own manner.

16. Swapping the Volume Manager Unit In and Out of Memory

When an application program that uses the volume manager unit is loaded,

COPYRIGHT 1984 APPLE COMPUTER, INC

the initialization code for the unit is executed. This code will set up VALID_DRIVES and some internal variables used by the volume manager unit. To conserve space in an application, this unit can be NOLOADed so that it is resident only when required. If this is done a call to INIT_VM must be made prior to using any other functions in the volume manager unit. This call will set up these variables again.

COPYRIGHT 1984 APPLE COMPUTER, INC