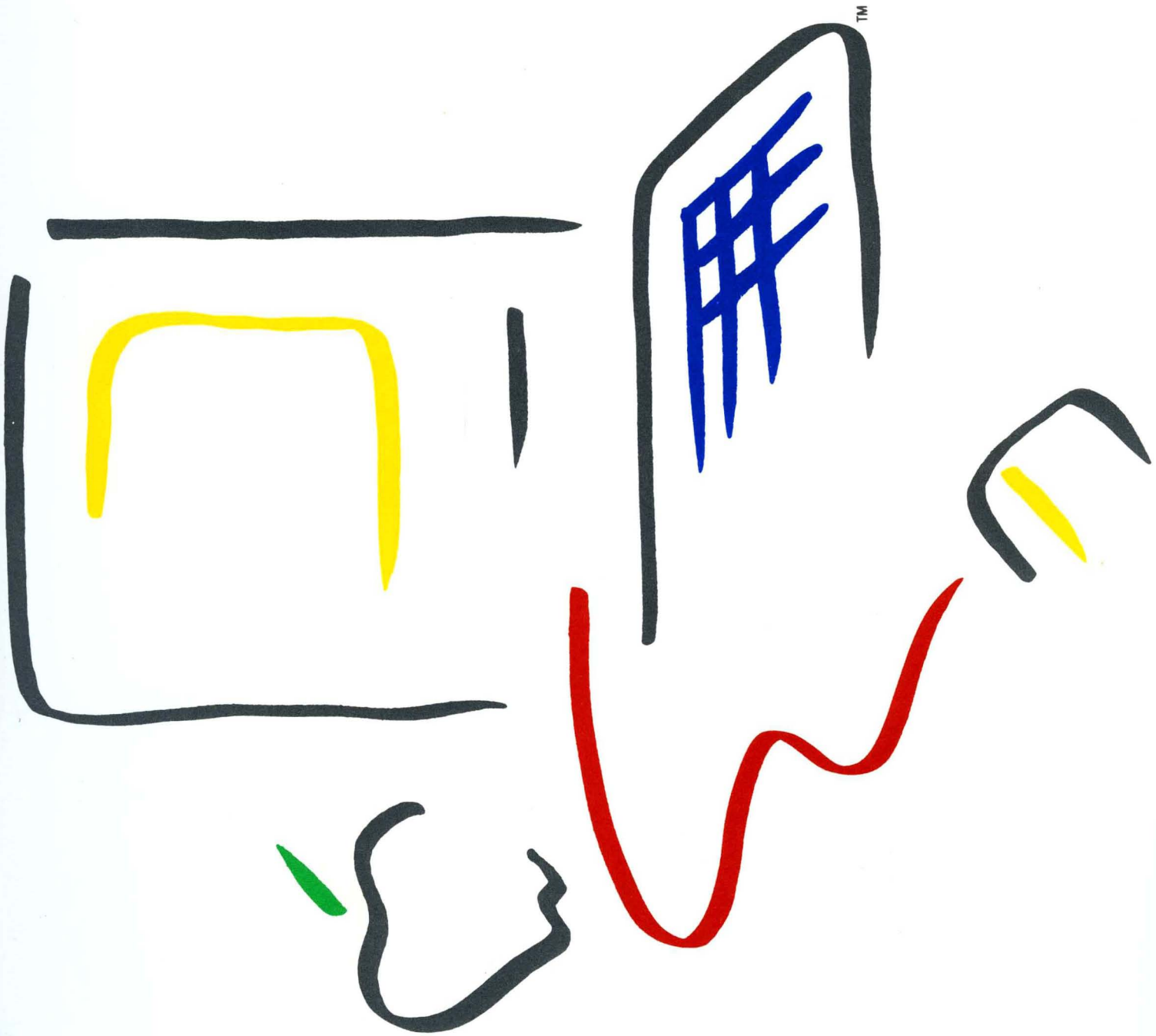


Macintosh™

MACINTOSH USER EDUCATION

Macintosh Packages: A Programmer's Guide /PACKAGES/PACK

See Also: The Macintosh User Interface Guidelines
The Memory Manager: A Programmer's Guide
QuickDraw: A Programmer's Guide
The Resource Manager: A Programmer's Guide
The Window Manager: A Programmer's Guide
Macintosh Control Manager Programmer's Guide
The Event Manager: A Programmer's Guide
The Dialog Manager: A Programmer's Guide
TextEdit: A Programmer's Guide
Programming Macintosh Applications in Assembly Language
The Structure of a Macintosh Application

Modification History: First Draft (ROM 7) B. Hacker & C. Rose 2/29/84
Second Draft Caroline Rose 5/7/84

ABSTRACT

Packages are sets of data structures and routines that are stored as resources and brought into memory only when needed. There's a package for presenting the standard user interface when a file is to be saved or opened, and others for doing more specialized operations such as floating-point arithmetic. This manual describes packages and the Package Manager, the part of the Macintosh User Interface Toolbox that provides access to packages.

Summary of significant changes and additions since last draft:

- The documentation of the International Utilities Package and the Binary-Decimal Conversion Package has been added.

- There's a new feature in the Standard File Package routine SFGetFile, whereby the user can select a file name by pressing a key.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes packages and the Package Manager. The Macintosh packages include one for presenting the standard user interface when a file is to be saved or opened, and others for doing more specialzed operations such as floating-point arithmetic. The Package Manager is the part of the Macintosh User Interface Toolbox that provides access to packages. *** Eventually, this will become part of the comprehensive Inside Macintosh manual. ***

You should already be familiar with the Macintosh User Interface Guidelines, Lisa Pascal, the Macintosh Operating System's Memory Manager, and the Resource Manager. Using the various packages may require that you be familiar with other parts of the Toolbox and Operating System as well.

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly-language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it.

The manual begins with a discussion of the Package Manager and packages in general. This is followed by a series of sections on the individual packages. You'll only need to read the sections about the packages that interest you. Each section describes the package briefly, tells how its routines fit into the flow of your application program, and then gives detailed descriptions of the package's routines.

Finally, there are summaries of the Package Manager and the individual packages, for quick reference, followed by a glossary of terms used in this manual.

THE PACKAGE MANAGER

The Package Manager is the part of the Macintosh User Interface Toolbox that enables you to access packages. Packages are sets of data structures and routines that are stored as resources and brought into memory only when needed. They serve as extensions to the Macintosh Operating System and User Interface Toolbox, for the most part performing less common operations.

The Macintosh packages, which are stored in the system resource file, include the following:

- The Standard File Package, for presenting the standard user interface when a file is to be saved or opened.

- The Disk Initialization Package, for initializing and naming new disks. This package is called by the Standard File Package; you'll only need to call it in nonstandard situations.

- The International Utilities Package, for accessing country-dependent information such as the formats for numbers, currency, dates, and times.

- The Binary-Decimal Conversion Package, for converting integers to decimal strings and vice versa.

- The Floating-Point Arithmetic and Transcendental Functions Packages. *** These packages, which occupy a total of about 8.5K bytes, will be documented in a future draft of this manual. ***

Packages have the resource type 'PACK' and the following resource IDs:

```
CONST dskInit = 2;  {Disk Initialization}
      stdFile = 3;  {Standard File}
      flPoint = 4;  {Floating-Point Arithmetic}
      trFunc  = 5;  {Transcendental Functions}
      intUtil = 6;  {International Utilities}
      bdConv  = 7;  {Binary-Decimal Conversion}
```

Assembly-language note: All macros for calling the routines in a particular package expand to invoke one macro, _PackN, where N is the resource ID of the package. The package determines which routine to execute from the routine selector, an integer that's passed to it on the stack. For example, the routine selector for the Standard File Package procedure SFPutFile is 1, so invoking the macro _SFPutFile pushes 1 onto the stack and invokes _Pack3.

There are two Package Manager routines that you can call directly from Pascal: one that lets you access a specified package and one that lets

you access all packages. The latter will already have been called when your application starts up, so normally you won't ever have to call the Package Manager yourself. Its procedures are described below for advanced programmers who may want to use them in unusual situations.

```
PROCEDURE InitPack (packID: INTEGER);
```

InitPack enables you to use the package specified by packID, which is the package's resource ID. (It gets a handle that will be used later to read the package into memory.)

```
PROCEDURE InitAllPacks;
```

InitAllPacks enables you to use all Macintosh packages (as though InitPack were called for each one). It will already have been called when your application starts up.

THE INTERNATIONAL UTILITIES PACKAGE

The International Utilities Package contains routines and data types that enable you to make your Macintosh application country-independent. Routines are provided for formatting dates and times and comparing strings in a way that's appropriate to the country where your application is being used. There's also a routine for testing whether to use the metric system of measurement. These routines access country-dependent information (stored in a resource file) that also tells how to format numbers and currency; you can access this information yourself for your own routines that may require it.

*** In the Inside Macintosh manual, the documentation of this package will be at the end of the volume that describes the User Interface Toolbox. ***

You should already be familiar with the Resource Manager, the Package Manager, and packages in general.

International Resources

Country-dependent information is kept in the system resource file in two resources of type 'INTL', with the resource IDs Ø and 1:

- International resource Ø contains the format for numbers, currency, and time, a short date format, and an indication of whether to use the metric system.

- International resource 1 contains a longer format for dates (spelling out the month and possibly the day of the week, with or without abbreviation) and a routine for localizing string comparison.

The system resource file released in each country contains the standard international resources for that country. Figure I-1 illustrates the standard formats for the United States, Great Britain, Italy, Germany, and France.

	United States	Great Britain	Italy	Germany	France
Numbers	1,234.56	1,234.56	1.234,56	1.234,56	1 234.56
List seperator	;	,	;	;	;
Currency	$0.23	£0.23	L. 0,23	0,23 DM	0,23 F
	($0.45)	(£0.45)	L. -0,45	-0,45 DM	-0,45 F
	$345.00	£345	L. 345	325,00 DM	325 F
Time	9:05 AM	09:05	9:05	9.05 Uhr	9:05
	11:30 AM	11:30	11:30	11.30 Uhr	11:30
	11:20 PM	23:20	23:20	23.20 Uhr	23:20
	11:20:09 PM	23:20:00	23:20:09	23.20.09 Uhr	23:20:09
Short date	12/22/84	22/12/1984	22-12-1984	22.12.1984	22.12.84
	2/ 1/84	01/02/1984	1-02-1984	1.02.1984	1.02.84

		Unabbreviated	Abbreviated
Long date	United States	Wednesday, February 1, 1984	Wed, Feb 1, 1984
	Great Britain	Wednesday, February 1, 1984	Wed, Feb 1, 1984
	Italy	mercoledì 1 Febbraio 1984	mer 1 Feb 1984
	Germany	Mittwoch, 1. Februar 1984	Mit, 1. Feb 1984
	France	Mercredi 1 fevrier 1984	Mer 1 fev 1984

Figure I-1. Standard International Formats

The routines in the International Utilities Package use the information in these resources; for example, the routines for formatting dates and times yield strings that look like those shown in Figure I-1. Routines in other packages, in desk accessories, and in ROM also access the international resources when necessary, as should your own routines if they need such information.

In some cases it may be appropriate to store either or both of the international resources in the application's or document's resource file, to override those in the system resource file. For example, suppose an application creates documents containing currency amounts and gets the currency format from international resource Ø. Documents created by such an application should have their own copy of the international resource Ø that was used to create them, so that the unit of currency will be the same if the document is displayed on a Macintosh configured for another country.

Information about the exact components and structure of each international resource follows here; you can skip this if you intend only to call the formatting routines in the International Utilities Package and won't access the resources directly yourself.

International Resource Ø

The International Utilities Package contains the following data types for accessing international resource Ø:

```
TYPE Int1ØHndl = ^Int1ØPtr;  *** Following "Int" is the letter "1" ***
     Int1ØPtr  = ^Int1ØRec;
     Int1ØRec  = PACKED RECORD
                   decimalPt:    CHAR; {decimal point character}
                   thousSep:     CHAR; {thousands separator}
                   listSep:      CHAR; {list separator}
                   currSym1:     CHAR; {currency symbol}
                   currSym2:     CHAR;
                   currSym3:     CHAR;
                   currFmt:      Byte; {currency format}
                   dateOrder:    Byte; {order of short date elements}
                   shortDateFmt: Byte; {short date format}
                   dateSep:      CHAR; {date separator}
                   timeCycle:    Byte; {Ø if 24-hour cycle, 255 if 12-hour}
                   timeFmt:      Byte; {time format}
                   mornStr:      PACKED ARRAY[1..4] OF CHAR;
                                  {trailing string for first 12-hour cycle}
                   eveStr:       PACKED ARRAY[1..4] OF CHAR;
                                  {trailing string for last 12-hour cycle}
                   timeSep:      CHAR; {time separator}
                   time1Suff:    CHAR; {trailing string for 24-hour cycle}
                   time2Suff:    CHAR;
                   time3Suff:    CHAR;
                   time4Suff:    CHAR;
                   time5Suff:    CHAR;
                   time6Suff:    CHAR;
                   time7Suff:    CHAR;
                   time8Suff:    CHAR;
                   metricSys:    Byte;   {255 if metric, Ø if not}
                   int1ØVers:    INTEGER {version information}
                 END;
```

(note)

A NUL character (ASCII code Ø) in a field of type CHAR means there's no such character. The currency symbol and the trailing string for the 24-hour cycle are separated into individual CHAR fields because of Pascal packing conventions. All strings include any required spaces.

The decimalPt, thousSep, and listSep fields define the number format. The thousands separator is the character that separates every three digits to the left of the decimal point. The list separator is the character that separates numbers, as when a list of numbers is entered by the user; it must be different from the decimal point character. If it's the same as the thousands separator, the user must not include the latter in entered numbers.

CurrSyml through currSym3 define the currency symbol (only one character for the United States and Great Britain, but two for France and three for Italy and Germany). CurrFmt determines the rest of the currency format, as shown in Figure I-2. The decimal point character and thousands separator for currency are the same as in the number format.

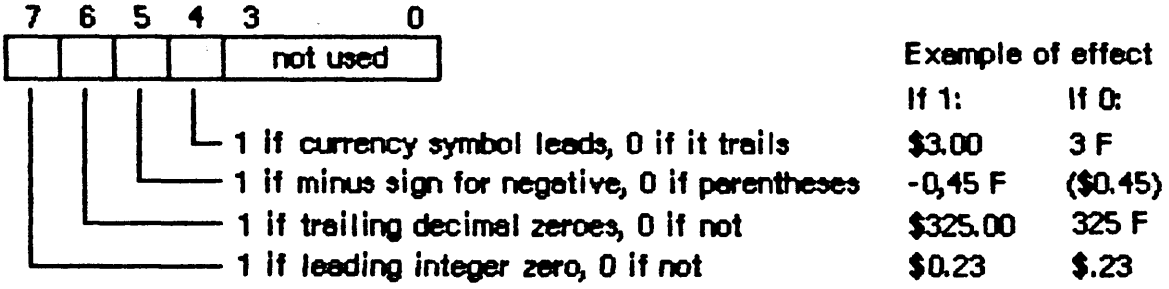


Figure I-2. CurrFmt Field

The following predefined constants are masks that can be used to set or test the bits in the currFmt field:

```
CONST currSymLead   = 16;  {set if currency symbol leads}
      currNegSym    = 32;  {set if minus sign for negative}
      currTrailingZ = 64;  {set if trailing decimal zeroes}
      currLeadingZ  = 128; {set if leading integer zero}
```

(note)
You can also apply the currency format's leading- and trailing-zero indicators to the number format if desired.

The dateOrder, shortDateFmt, and dateSep fields define the short date format. DateOrder indicates the order of the day, month, and year, with one of the following values:

```
CONST mdy = Ø;  {month day year}
      dmy = 1;  {day month year}
      ymd = 2;  {year month day}
```

ShortDateFmt determines whether to show leading zeroes in day and month numbers and whether to show the century, as illustrated in Figure I-3. DateSep is the character that separates the different parts of the date.

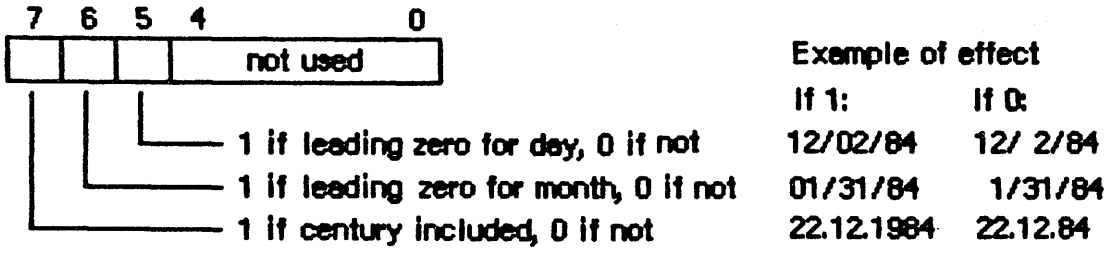


Figure I-3. ShortDateFmt Field

To set or test the bits in the shortDateFmt field, you can use the following predefined constants as masks:

```
CONST dayLeadingZ = 32;  {set if leading zero for day}
      mntLeadingZ = 64;  {set if leading zero for month}
      century     = 128; {set if century included}
```

The next several fields define the time format: the cycle (12 or 24 hours); whether to show leading zeroes (timeFmt, as shown in Figure I-4); a string to follow the time (two for 12-hour cycle, one for 24-hour); and the time separator character.

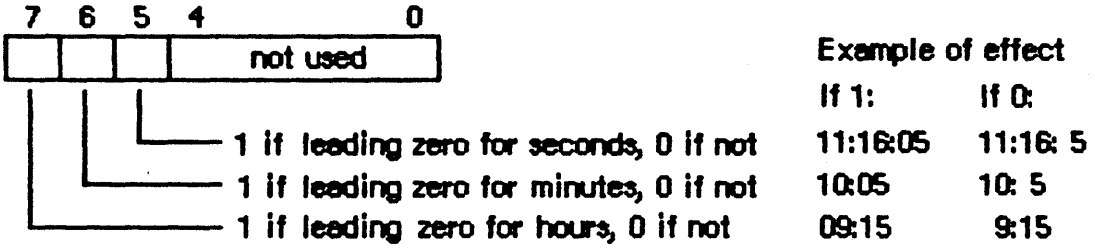


Figure I-4. TimeFmt Field

The following masks are available for setting or testing bits in the timeFmt field:

```
CONST secLeadingZ = 32;  {set if leading zero for seconds}
      minLeadingZ = 64;  {set if leading zero for minutes}
      hrLeadingZ  = 128; {set if leading zero for hours}
```

MetricSys indicates whether to use the metric system. The last field, int1ØVers, contains a version number in its low-order byte and one of the following constants in its high-order byte:

```
CONST verUS      = Ø;
      verFrance  = 1;
      verBritain = 2;
      verGermany = 3;
      verItaly   = 4;
```

International Resource 1

The International Utilities Package contains the following data types for accessing international resource 1:

```
TYPE Intl1Hndl = ^Intl1Ptr; *** Following "Int" is the letter "l" ***
     Intl1Ptr  = ^Intl1Rec; *** Following "Intl" is the number "l" ***
     Intl1Rec  = PACKED RECORD
                   days:         ARRAY[1..7] OF STRING[15];  {day names}
                   months:       ARRAY[1..12] OF STRING[15]; {month names}
                   suppressDay:  Byte;   {Ø for day name, 255 for none}
                   longDateFmt:  Byte;   {order of long date elements}
                   dayleadingØ:  Byte;   {255 for leading Ø in day number}
                   abbrLen:      Byte;   {length for abbreviating names}
                   stØ:          PACKED ARRAY[1..4] OF CHAR; {strings }
                   st1:          PACKED ARRAY[1..4] OF CHAR; { for }
                   st2:          PACKED ARRAY[1..4] OF CHAR; { long }
                   st3:          PACKED ARRAY[1..4] OF CHAR; { date }
                   st4:          PACKED ARRAY[1..4] OF CHAR; { format}
                   intl1Vers:    INTEGER; {version information}
                   localRtn:     INTEGER  {routine for localizing string }
                                          { comparison; actually may be }
                                          { longer than one integer}
                 END;
```

All fields except the last two determine the long date format. The day names in the days array are ordered from Sunday to Saturday. (The month names are of course ordered from January to December.) As shown below, the longDateFmt field determines the order of the various parts of the date. StØ through st4 are strings (usually punctuation) that appear in the date.

longDateFmt	Long date format								
Ø	stØ	day name	st1	day	st2	month	st3	year	st4
255	stØ	day name	st1	month	st2	day	st3	year	st4

See Figure I-5 for examples of how the International Utilities Package formats dates based on these fields. The examples assume that the suppressDay and dayLeadingØ fields contain Ø. A suppressDay value of 255 causes the day name and st1 to be omitted, and a dayLeading value of 255 causes a Ø to appear before day numbers less than 1Ø.

longDateFmt	st0	st1	st2	st3	st4	Sample result
0	"	', '	'. '	' '	"	Mittwoch, 2. Februar 1984
255	"	', '	' '	', '	"	Wednesday, February 1, 1984

Figure I-5. Long Date Formats

AbbrLen is the number of characters to which month and day names should be abbreviated when abbreviation is desired.

The intl1Vers field contains version information with the same format as the intlØVers field of international resource Ø.

LocalRtn contains a routine that localizes the built-in character ordering (as described below under "International String Comparison").

International String Comparison

The International Utilities Package lets you compare strings in a way that accounts for diacritical marks and other special characters. The sort order built into the package may be localized through a routine stored in internatiomal resource 1.

The sort order is determined by a ranking of the entire Macintosh character set. The ranking can be thought of as a two-dimensional table. Each row of the table is a class of characters such as all A's (uppercase and lowercase, with and without diacritical marks). The characters are ordered within each row, but this ordering is secondary to the order of the rows themselves. For example, given that the rows for letters are ordered alphabetically, the following are all true under this scheme:

```
      'A' < 'a'
and   'Ab' < 'ab'
but   'Ac' > 'ab'
```

Even though 'A' < 'a' within the A row, 'Ac' > 'ab' because the order 'c' > 'b' takes precedence over the secondary ordering of the 'a' and the 'A'. In effect, the secondary ordering is ignored unless the comparison based on the primary ordering yields equality.

(note)
The Pascal relational operators are used here for convenience only. String comparison in Pascal yields very different results, since it simply follows the ordering of the characters' ASCII codes.

When the strings being compared are of different lengths, each character in the longer string that doesn't correspond to a character in the shorter one compares "greater"; thus 'a' < 'ab'. This takes precedence over secondary ordering, so 'a' < 'Ab' even though 'A' < 'a'.

Besides letting you compare strings as described above, the International Utilities Package includes a routine that compares strings for equality without regard for secondary ordering. The effect on comparing letters, for example, is that diacritical marks are ignored and uppercase and lowercase are not distinguished.

Figure I-6 on the following page shows the two-dimensional ordering of the character set (from least to greatest as you read from top to bottom or left to right). The numbers on the left are ASCII codes corresponding to each row; ellipses (...) designate sequences of rows of just one character. Some codes do not correspond to rows (such as $61 through $7A, because lowercase letters are included in with their uppercase equivalents). See the Toolbox Event Manager manual for a table showing all the characters and their ASCII codes.

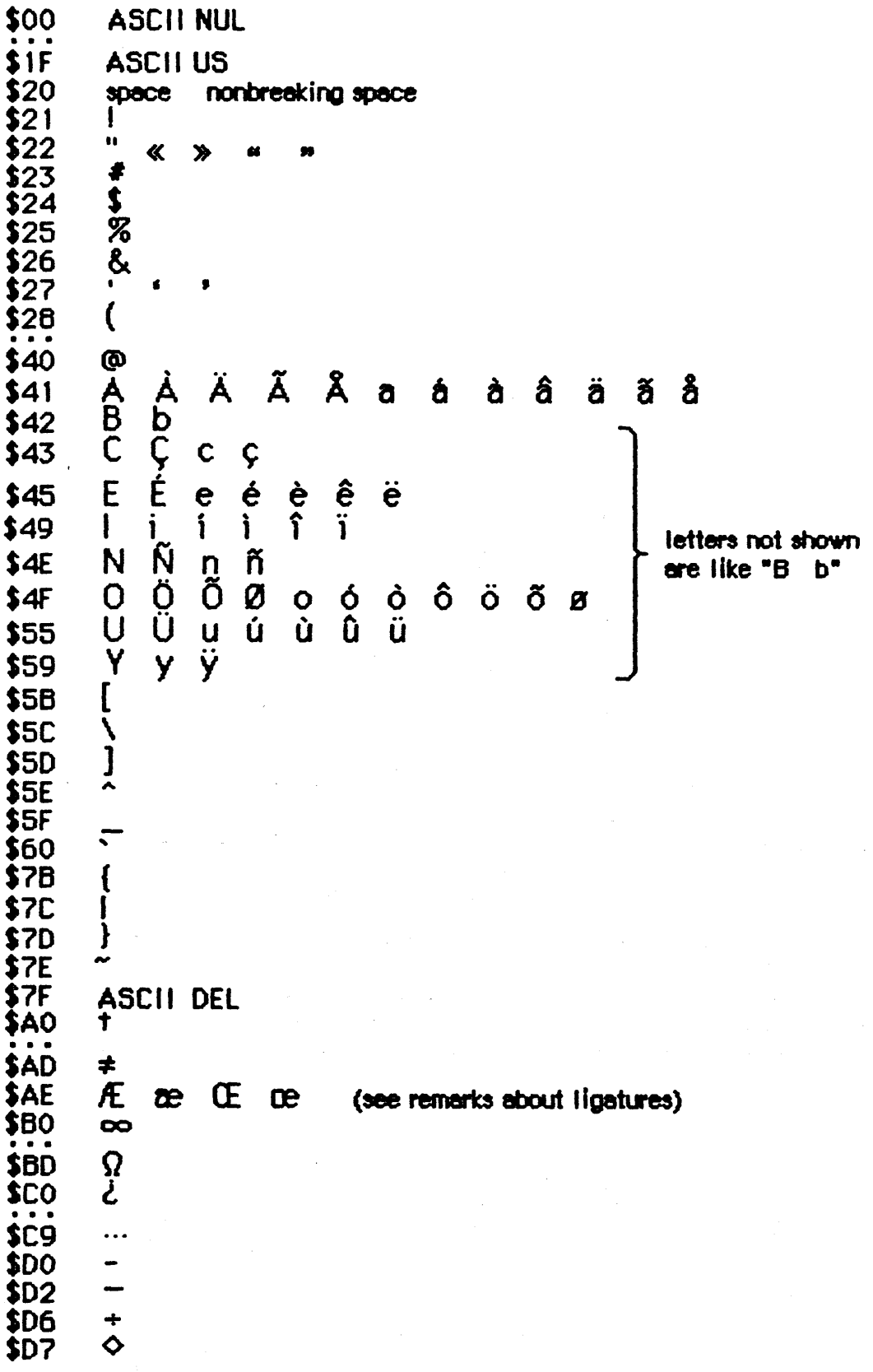


Figure I-6. International Character Ordering

Characters combining two letters, as in the $AE row, are called ligatures. As shown in Figure I-7, they're actually expanded to the corresponding two letters, in the following sense:

- Primary ordering: The ligature is equal to the two-character sequence.
- Secondary ordering: The ligature is greater than the two-character sequence.

Figure I-7. Ordering for Special Characters

Ligatures are ordered somewhat differently in Germany to accommodate umlauted characters (see Figure I-7). This is accomplished by means of the routine in international resource 1 for localizing the built-in character ordering. In the system resource file for Germany, this routine expands umlauted characters to the corresponding two letters (for example, "AE" for A-umlaut). The secondary ordering places the umlauted character between the two-character sequence and the ligature, if any. Likewise, the German double-s character expands to "ss".

In the system resource file for Great Britain, the localization routine in international resource 1 orders the pound currency sign between double quote and the pound weight sign (see Figure I-8). For the United States, France, and Italy, the localization routine does nothing.

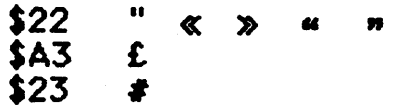


Figure I-8. Special Ordering for Great Britain

Assembly-language note: The null localization routine consists of an RTS instruction.

*** Information on how to write your own localization routine is forthcoming. ***

Using the International Utilities Package

This section discusses how the routines in the International Utilities package fit into the general flow of an application program, and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

The International Utilities Package is automatically read into memory from the system resource file when one of its routines is called. When a routine needs to access an international resource, it asks the Resource Manager to read the resource into memory. Together, the package and its resources occupy about 2K bytes.

As described in the *** not yet existing *** Operating System Utilities manual, you can get the date and time as a long integer from the utility routine ReadDateTime. If you need a string corresponding to the date or time, you can pass this long integer to the IUDateString or IUTimeString procedure in the International Utilities Package. These procedures determine the local format from the international resources read into memory by the Resource Manager (that is, resource type 'INTL' and resource ID Ø or 1). In some situations, you may need the format information to come instead from an international resource that you specify by its handle; if so, you can use IUDatePString or IUTimePString. This is useful, for example, if you want to use an international resource in a document's resource file after you've closed that file.

Applications that use measurements, such as on a ruler for setting margins and tabs, can call IUMetric to find out whether to use the metric system. This function simply returns the value of the corresponding field in international resource Ø. To access any other fields in an international resource--say, the currency format in international resource Ø--call IUGetIntl to get a handle to the resource. If you change any of the fields and want to write the changed resource to a resource file, the IUSetIntl procedure lets you do this.

To sort strings, you can use IUCompString or, if you're not dealing with Pascal strings, the more general IUMagString. These routines compare two strings and give their exact relationship, whether equal, less than, or greater than. Subtleties like diacritical marks and case differences are taken into consideration, as described above under "International String Comparison". If you need to know only whether two strings are equal, and want to ignore the subtleties, use IUEqualString (or the more general IUMagIDString) instead.

(note)

The Operating System utility routine EqualString also compares two Pascal strings for equality. It's less sophisticated than IUEqualString in that it follows ASCII

order more strictly; for details, see the Operating System Utilities manual *** eventually ***.

International Utilities Package Routines

Assembly-language note: The macros for calling the International Utilities Package routines push one of the following routine selectors onto the stack and then invoke _Pack6:

Routine	Selector
IUDatePString	14
IUDateString	Ø
IUGetIntl	6
IUMagIDString	12
IUMagString	1Ø
IUMetric	4
IUTimePString	16
IUTimeString	2
IUSetIntl	8

```
PROCEDURE IUDateString (dateTime: LongInt; form: DateForm; VAR result:
          Str255);
```

Given a date and time as returned by the Operating System Utility routine ReadDateTime, IUDateString returns in the result parameter a string that represents the corresponding date. The form parameter has the following data type:

```
TYPE DateForm = (shortDate, longDate, abbrevDate);
```

ShortDate requests the short date format, longDate the long date, and abbrevDate the abbreviated long date. IUDateString determines the exact format from international resource Ø for the short date or 1 for the long date. See Figure I-1 above for examples of the standard formats. Notice that the short date contains a space in place of a leading zero when the format specifies "no leading zero", so the length of the result is always the same for short dates.

If the abbreviated long date is requested and the abbreviation length in international resource 1 is greater than the actual length of the name being abbreviated, IUDateString fills the abbreviation with NUL characters; the abbreviation length should not be greater than 15, the maximum name length.

```
PROCEDURE IUDatePString (dateTime: LongInt; form: DateForm; VAR result:
          Str255; intlParam: Handle);
```

IUDatePString is the same as IUDateString except that it determines the exact format of the date from the resource whose handle is passed in intlParam, overriding the resource that would otherwise be used.

```
PROCEDURE IUTimeString (dateTime: LongInt; wantSeconds: BOOLEAN; VAR
          result: Str255);
```

Given a date and time as returned by the Operating System Utility routine ReadDateTime, IUTimeString returns in the result parameter a string that represents the corresponding time of day. If wantSeconds is TRUE, seconds are included in the time; otherwise, only the hour and minute are included. IUTimeString determines the time format from international resource Ø. See Figure I-1 above for examples of the standard formats. Notice that the time contains a space in place of a leading zero when the format specifies "no leading zero", so the length of the result is always the same.

```
PROCEDURE IUTimePString (dateTime: LongInt; wantSeconds: BOOLEAN; VAR
          result: Str255; intlParam: Handle);
```

IUTimePString is the same as IUTimeString except that it determines the time format from the resource whose handle is passed in intlParam, overriding the resource that would otherwise be used.

```
FUNCTION IUMetric : BOOLEAN;
```

If international resource Ø specifies that the metric system is to be used, IUMetric returns TRUE; otherwise, it returns FALSE.

```
FUNCTION IUGetIntl (theID: INTEGER) : Handle;
```

IUGetIntl returns a handle to the international resource numbered theID (Ø or 1). It calls the Resource Manager function GetResource('INTL',theID). For example, if you want to access individual fields of international resource Ø, you can do the following:

```
VAR myHndl: Handle;
    intØ: IntlØHndl;
...
myHndl := IUGetIntl(Ø);
intØ := POINTER(ORD(myHndl));
```

```
PROCEDURE IUSetIntl (refNum: INTEGER; theID: INTEGER; intlParam:
          Handle);
```

In the resource file having the reference number refNum, IUSetIntl sets the international resource numbered theID (Ø or 1) to the data pointed to by intlParam. The data may be either an existing resource or data that hasn't yet been written to a resource file. IUSetIntl adds the resource to the specified file or replaces the resource if it's already there.

```
FUNCTION IUCompString (aStr,bStr: Str255) : INTEGER;  [Pascal only]
```

IUCompString compares aStr and bStr as described above under "International String Comparison", taking both primary and secondary ordering into consideration. It returns one of the values listed below.

Result	Meaning	Example	
		aStr	bStr
-1	aStr is less than bStr	'Ab'	'ab'
Ø	aStr equals bStr	'Ab'	'Ab'
1	aStr is greater than bStr	'Ac'	'ab'

Assembly-language note: IUCompString was created for the convenience of Pascal programmers; there's no trap for it. It eventually calls IUMagString, which is what you should use from assembly language.

```
FUNCTION IUMagString (aPtr,bPtr: Ptr; aLen,bLen: INTEGER) : INTEGER;
```

IUMagString is the same as IUCompString (above) except that instead of comparing two Pascal strings, it compares the string defined by aPtr and aLen to the string defined by bPtr and bLen. The pointer points to the first character of the string (any byte in memory, not necessarily word-aligned), and the length specifies the number of characters in the string.

```
FUNCTION IUEqualString (aStr,bStr: Str255) : INTEGER;  [Pascal only]
```

IUEqualString compares aStr and bStr for equality without regard for secondary ordering, as described above under "International String Comparison". If the strings are equal, it returns Ø; otherwise, it returns 1. For example, if the strings are 'Rose' and 'rose', IUEqualString considers them equal and returns Ø.

(note)
See also EqualString in the Operating System Utilities manual *** doesn't yet exist ***.

Assembly-language note: IUEqualString was created for the convenience of Pascal programmers; there's no trap for it. It eventually calls IUMagIDString, which is what you should use from assembly language.

FUNCTION IUMagIDString (aPtr,bPtr: Ptr; aLen,bLen: INTEGER) : INTEGER;

IUMagIDString is the same as IUEqualString (above) except that instead of comparing two Pascal strings, it compares the string defined by aPtr and aLen to the string defined by bPtr and bLen. The pointer points to the first character of the string (any byte in memory, not necessarily word-aligned), and the length specifies the number of characters in the string.

THE BINARY-DECIMAL CONVERSION PACKAGE

The Binary-Decimal Conversion Package contains only two routines: one converts an integer from its internal (binary) form to a string that represents its decimal (base 10) value; the other converts a decimal string to the corresponding integer.

*** In the Inside Macintosh manual, the documentation of this package will be at the end of the volume that describes the User Interface Toolbox. ***

You should already be familiar with the Package Manager, and packages in general.

The Binary-Decimal Conversion Package is automatically read into memory when one of its routines is called; it occupies a total of about 200 bytes. The routines are described below. They're register-based, so the Pascal form of each is followed by a box containing information needed to use the routine from assembly language. (For general information on using assembly language, see Programming Macintosh Applications in Assembly Language.)

Assembly-language note: The macros for calling the Binary-Decimal Conversion Package routines push one of the following routine selectors onto the stack and then invoke _Pack7:

Routine	Selector
NumToString	0
StringToNum	1

```
PROCEDURE NumToString (theNum: LongInt; VAR theString: Str255);
```

Trap macro	_NumToString	
On entry	A0:	pointer to theString (length byte followed by characters)
	D0:	theNum (long integer)
On exit	A0:	pointer to theString

NumToString converts theNum to a string that represents its decimal value, and returns the result in theString. If the value is negative, the string begins with a minus sign; otherwise, the sign is omitted. Leading zeroes are suppressed, except that the value 0 produces '0'.

For example:

theNum	theString
12	'12'
-23	'-23'
Ø	'Ø'

PROCEDURE StringToNum (theString: Str255; VAR theNum: LongInt);

Trap macro	_StringToNum
On entry	AØ: pointer to theString (length byte followed by characters)
On exit	DØ: theNum (long integer)

Given a string representing a decimal integer, StringToNum converts it to the corresponding integer and returns the result in theNum. The string may begin with a plus or minus sign. For example:

theString	theNum
'12'	12
'-23'	-23
'-Ø'	Ø
'Ø55'	55

The magnitude of the integer is converted modulo 2^32, and the 32-bit result is negated if the string begins with a minus sign; integer overflow occurs if the magnitude is greater than 2^31-1. (Negation is done by taking the two's complement--reversing the state of each bit and then adding 1.) For example:

theString		theNum
'2147483648'	(magnitude is 2^31)	-2147483648
'-2147483648'		-2147483648
'4294967295'	(magnitude is 2^32-1)	-1
'-4294967295'		1

StringToNum doesn't actually check whether the characters in the string are between 'Ø' and '9'; instead, since the ASCII codes for 'Ø' through '9' are $3Ø through $39, it just masks off the last four bits and uses them as a digit. For example, '2:' is converted to the number 3Ø because the ASCII code for ':' is $3A. Leading spaces before the first digit are treated as zeroes, since the ASCII code for a space is $2Ø. Given that the ASCII codes for 'C', 'A', and 'T' are $43, $41, and $54, respectively, consider the following examples:

theString	theNum
'CAT'	314
'+CAT'	314
'-CAT'	-314

THE STANDARD FILE PACKAGE

The Standard File Package provides the standard user interface for specifying a file to be saved or opened. It allows the file to be on a disk in any drive connected to the Macintosh, and lets a currently inserted disk be ejected so that another one can be inserted.

*** In the Inside Macintosh manual, the documentation of this package will be at the end of the volume that describes the Toolbox. ***

You should already be familiar with the following:

- the basic concepts and structures behind QuickDraw, particularly points and rectangles
- the Toolbox Event Manager
- the Dialog Manager, especially the ModalDialog procedure
- the Package Manager and packages in general

About the Standard File Package

Standard Macintosh applications should have a File menu from which the user can save and open documents, via the Save, Save As, and Open commands. In response to these commands, the application can call the Standard File Package to find out the document name and let the user switch disks if desired. As described below, a dialog box is presented for this purpose. (More details and illustrations are given later in the descriptions of the individual routines.)

When the user chooses Save As, or Save when the document is untitled, the application needs a name for the document. The corresponding dialog box lets the user enter the document name and click a button labeled "Save" (or just click "Cancel" to abort the command). By convention, the dialog box comes up displaying the current document name, if any, so the user can edit it.

In response to an Open command, the application needs to know which document to open. The corresponding dialog box displays the names of all documents that might be opened, and the user chooses one by clicking it and then clicking a button labeled "Open". A vertical scroll bar allows scrolling through the names if there are more than can be shown at once.

Both of these dialog boxes let the user:

- insert a disk in an external drive connected to the Macintosh
- eject a disk from either drive and insert another

- initialize and name an inserted disk that's uninitialized

- switch from one drive to another

On the right in the dialog box, separated from the rest of the box by a gray line, there's a disk name with one or two buttons below it; Figure S-1 shows what this looks like when an external drive is connected to the Macintosh but currently has no disk in it. Notice that the Drive button is inactive (dimmed). After the user inserts a disk in the external drive (and, if necessary, initializes and names it), the Drive button becomes active. If there's no external drive, the Drive button isn't displayed at all.

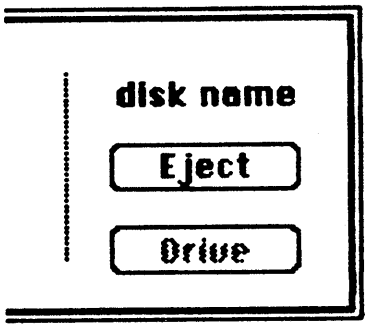


Figure S-1. Partial Dialog Box

The disk name displayed in the dialog box is the name of the current disk, initially the disk you used to start up the Macintosh. The user can click Eject to eject the current disk and insert another, which then becomes the current disk. If there's an external drive, clicking the Drive button changes the current disk from the one in the external drive to the one in the internal drive or vice versa. The Drive button is inactive whenever there's only one disk inserted.

If an uninitialized or otherwise unreadable disk is inserted, the Standard File Package calls the Disk Initialization Package to provide the standard user interface for initializing and naming a disk.

Using the Standard File Package

This section discusses how the routines in the Standard File Package fit into the general flow of an application program, and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

The Standard File Package and the resources it uses are automatically read into memory when one of its routines is called. It in turn reads the Disk Initialization Package into memory if a disk is ejected; together they occupy about 6.5K bytes.

Call SFPutFile when your application is to save to a file and needs to get the name of the file from the user. Standard applications should do this when the user chooses Save As from the File menu, or Save when the document is untitled. SFPutFile displays a dialog box allowing the

user to enter a file name.

Similarly, SFGetFile is useful whenever your application is to open a file and needs to know which one, such as when the user chooses the Open command from a standard application's File menu. SFGetFile displays a dialog box with a list of file names to choose from.

You pass these routines a reply record, as shown below, and they fill it with information about the user's reply.

```
TYPE SFReply = RECORD
                 good:    BOOLEAN;     {FALSE if ignore command}
                 copy:    BOOLEAN;     {not used}
                 fType:   OSType;      {file type or not used}
                 vRefNum: INTEGER;     {volume reference number}
                 version: INTEGER;     {file's version number}
                 fName:   STRING[63]   {file name}
               END;
```

The first field of this record determines whether the file operation should take place or the command should be ignored (because the user clicked the Cancel button in the dialog box). The fType field is used by SFGetFile to store the file's type. The vRefNum, version, and fName fields identify the file chosen by the user; the application passes their values on to the File Manager routine that does the actual file operation. VRefNum contains the volume reference number of the volume containing the file. Currently the version field always contains Ø; the use of nonzero version numbers is **not** supported by this package. For more information on files, volumes, and file operations, see the File Manager manual *** doesn't yet exist ***.

Both SFPutFile and SFGetFile allow you to use a nonstandard dialog box; two additional routines, SFPPutFile and SFPGetFile, provide an even more convenient and powerful way of doing this.

Standard File Package Routines

Assembly-language note: The macros for calling the Standard File Package routines push one of the following routine selectors onto the stack and then invoke _Pack3:

Routine	Selector
SFGetFile	2
SFPGetFile	4
SFPPutFile	3
SFPutFile	1

```
PROCEDURE SFPutFile (where: Point; prompt: Str255; origName: Str255;
          dlgHook: ProcPtr; VAR reply: SFReply);
```

SFPutFile displays a dialog box allowing the user to specify a file to which data will be written (as during a Save or Save As command). It then repeatedly gets and handles events until the user either confirms the command after entering an appropriate file name or aborts the command by clicking Cancel in the dialog. It reports the user's reply by filling the fields of the reply record specified by the reply parameter, as described above; the fType field of this record isn't used.

The general appearance of the standard SFPutFile dialog box is shown in Figure S-2. The where parameter specifies the location of the top left corner of the dialog box in global coordinates. The prompt parameter is a line of text to be displayed as a statText item in the dialog box, where shown in Figure S-2. The origName parameter contains text that appears as an enabled, selected editText item; for the standard document-saving commands, it should be the current name of the document, or the empty string (to display an insertion point) if the document hasn't been named yet.

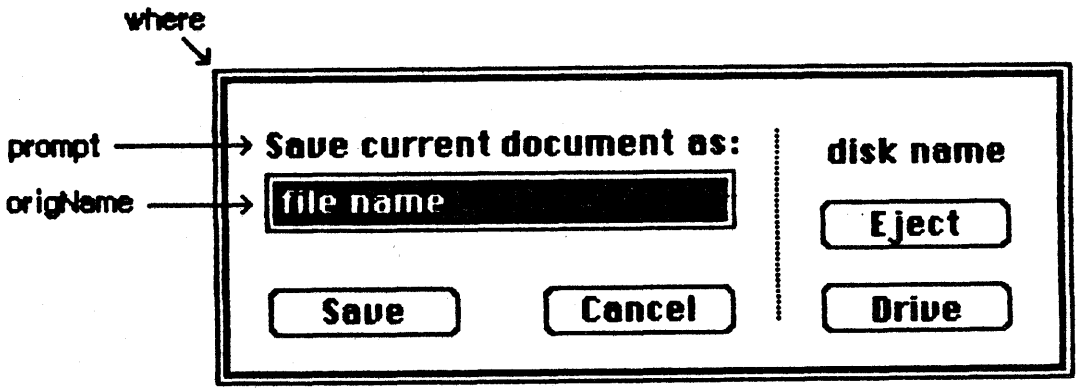


Figure S-2. Standard SFPutFile Dialog

If you want to use the standard SFPutFile dialog box, pass NIL for dlgHook; otherwise, see the information for advanced programmers below.

SFPutFile repeatedly calls the Dialog Manager procedure ModalDialog. When an event involving an enabled dialog item occurs, ModalDialog handles the event and returns the item number, and SFPutFile responds as follows:

- If the Eject or Drive button is clicked, or a disk is inserted, SFPutFile responds as described above under "About the Standard File Package".

- Text entered into the editText item is stored in the fName field of the reply record. (SFPutFile keeps track of whether there's currently any text in the item, and makes the Save button inactive if not.)

- If the Save button is clicked, SFPutFile determines whether the file name in the fName field of the reply record is appropriate. If so, it returns control to the application with the first field of the reply record set to TRUE; otherwise, it responds accordingly, as described below.

- If the Cancel button in the dialog is clicked, SFPutFile returns control to the application with the first field of the reply record set to FALSE.

(note)

Notice that disk insertion is one of the user actions listed above, even though ModalDialog normally ignores disk-inserted events. The reason this works is that SFPutFile calls ModalDialog with a filterProc function that checks for a disk-inserted event and returns a "fake", very large item number if one occurs; SFPutFile recognizes this item number as an indication that a disk was inserted.

The situations that may cause an entered name to be inappropriate, and SFPutFile's response to each, are as follows:

- If a file with the specified name already exists on the disk and is different from what was passed in the origName parameter, the alert in Figure S-3 is displayed. If the user clicks Yes, the file name is appropriate.

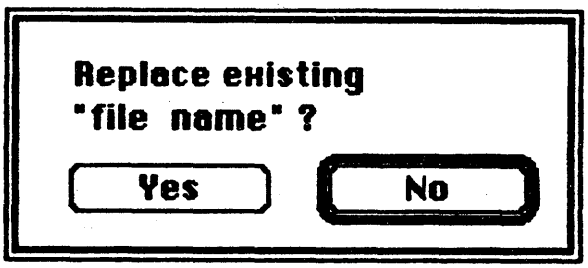


Figure S-3. Alert for Existing File

- If the disk to which the file should be written is locked, the alert in Figure S-4 is displayed. If a system error occurs, a similar alert is displayed, with a corresponding message explaining the problem.

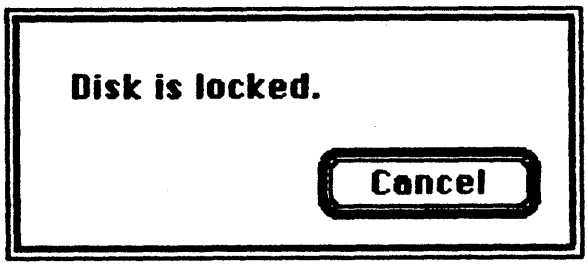


Figure S-4. Alert for Locked Disk

(note)
The user may specify a disk name (preceding the file name and separated from it by a colon). If the disk isn't currently in a drive, an alert similar to the one in Figure S-4 is displayed. The ability to specify a disk name is supported for historical reasons only; users should not be encouraged to do it.

After the user clicks No or Cancel in response to one of these alerts, SFPutFile dismisses the alert box and continues handling events (so a different name may be entered).

Advanced programmers: You can create your own dialog box rather than use the standard SFPutFile dialog. To do this, you must provide your own dialog template and store it in your application's resource file with the same resource ID that the standard template has in the system resource file:

```
CONST putDlgID = -3999;  {SFPutFile dialog template ID}
```

(note)
The SFPPutFile procedure, described below, lets you use any resource ID for your nonstandard dialog box.

Your dialog template must specify that the dialog window be invisible, and your dialog must contain all the standard items, as listed below. The appearance and location of these items in your dialog may be different. You can make an item "invisible" by giving it a display rectangle that's off the screen. The display rectangle for each item in the standard dialog box is given below. The rectangle for the standard dialog box itself is (Ø, Ø, 3Ø4, 1Ø4).

Item number	Item	Standard display rectangle
1	Save button	(12, 74, 82, 92)
2	Cancel button	(114, 74, 184, 92)
3	Prompt string (statText)	(12, 12, 184, 28)
4	UserItem for disk name	(2Ø9, 16, 295, 34)
5	Eject button	(217, 43, 287, 61)
6	Drive button	(217, 74, 287, 92)
7	EditText item for file name	(14, 34, 182, 5Ø)
8	UserItem for gray line	(2ØØ, 16, 2Ø1, 88)

(note)

Remember that the display rectangle for any "invisible" item must be at least about 2Ø pixels wide. *** This will be discussed in a future draft of the Dialog Manager manual. ***

If your dialog has additional items beyond the the standard ones, or if you want to handle any of the standard items in a nonstandard manner, you must write your own dlgHook function and point to it with dlgHook. Your dlgHook function should have two parameters and return an integer value. For example, this is how it would be declared if it were named MyDlg:

```
FUNCTION MyDlg (item: INTEGER; theDialog: DialogPtr) : INTEGER;
```

Immediately after calling ModalDialog, SFPutFile calls your dlgHook function, passing it the item number returned by ModalDialog and a pointer to the dialog record describing your dialog box. Using these two parameters, your dlgHook function should determine how to handle the event. There are predefined constants for the item numbers of standard enabled items, as follows:

```
CONST putSave   = 1;  {Save button}
      putCancel = 2;  {Cancel button}
      putEject  = 5;  {Eject button}
      putDrive  = 6;  {Drive button}
      putName   = 7;  {editText item for file name}
```

ModalDialog also returns the "fake" item number 1ØØ when a disk-inserted event occurs, as detected by its filterProc function.

After handling the event (or, perhaps, after ignoring it) the dlgHook function must return an item number to SFPutFile. If the item number is one of those listed above, SFPutFile responds in the standard way; otherwise, it does nothing.

(note)

For advanced programmers who want to change the appearance of the alerts displayed when an inappropriate file name is entered, the resource IDs of those alerts in the system resource file are listed below.

Alert	Resource ID
Existing file	-3996
Locked disk	-3997
System error	-3995
Disk not found	-3994

```
PROCEDURE SFPPutFile (where: Point; prompt: Str255; origName: Str255;
          dlgHook: ProcPtr; VAR reply: SFReply; dlgID: INTEGER;
          filterProc: ProcPtr);
```

SFPPutFile is an alternative to SFPutFile for advanced programmers who want to use a nonstandard dialog box. It's the same as SFPutFile except for the two additional parameters dlgID and filterProc.

DlgID is the resource ID of the dialog template to be used instead of the standard one (so you can use whatever ID you wish rather than the same one as the standard).

The filterProc parameter determines how ModalDialog will filter events when called by SFPPutFile. If filterProc is NIL, ModalDialog does the standard filtering that it does when called by SFPutFile; otherwise, filterProc should point to a function for ModalDialog to execute **after** doing the standard filtering. The function must be the same as one you'd pass directly to ModalDialog in its filterProc parameter. (See the Dialog Manager manual for more information.)

```
PROCEDURE SFGetFile (where: Point; prompt: Str255; fileFilter: ProcPtr;
          numTypes: INTEGER; typeList: SFTypeList; dlgHook: ProcPtr;
          VAR reply: SFReply);
```

SFGetFile displays a dialog box listing the names of a specific group of files from which the user can select one to be opened (as during an Open command). It then repeatedly gets and handles events until the user either confirms the command after choosing a file name or aborts the command by clicking Cancel in the dialog. It reports the user's reply by filling the fields of the reply record specified by the reply parameter, as described above under "Using the Standard File Package".

The general appearance of the standard SFGetFile dialog box is shown in Figure S-5. File names are sorted in order of the ASCII codes of their characters, ignoring diacritical marks and mapping lowercase characters to their uppercase equivalents. If there are more file names than can be displayed at one time, the scroll bar is active; otherwise, the scroll bar is inactive.

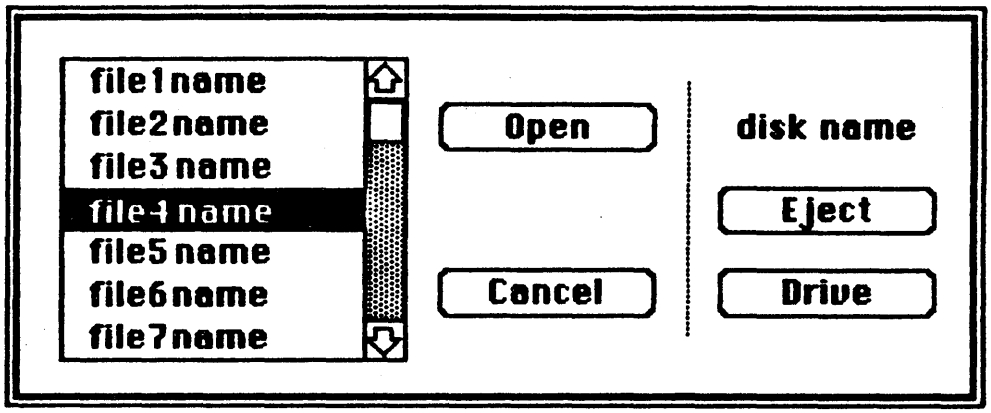


Figure S-5. Standard SFGetFile Dialog

The where parameter specifies the location of the top left corner of the dialog box in global coordinates. The prompt parameter is ignored; it's there for historical purposes only.

The fileFilter, numTypes, and typeList parameters determine which files appear in the dialog box. SFGetFile first looks at numTypes and typeList to determine what types of files to display, then it executes the function pointed to by fileFilter (if any) to do additional filtering on which files to display. File types are discussed in the manual The Structure of a Macintosh Application. For example, if the application is concerned only with pictures, you won't want to display the names of any text files.

Pass -1 for numTypes to display all types of files; otherwise, pass the number of file types you want to display, and pass the types themselves in typeList. The SFTypeList data type is defined as follows:

```
TYPE SFTypeList = ARRAY [Ø..3] OF OSType;
```

(note)
This array is declared for a reasonable maximum number of types (four). If you need to specify more than four types, declare your own array type with the desired number of entries (and use the @ operator to pass a pointer to it).

If fileFilter isn't NIL, SFGetFile executes the function it points to for each file, to determine whether the file should be displayed. The fileFilter function has one parameter and returns a Boolean value. For example:

```
FUNCTION MyFileFilter (paramBlock: ParmBlkPtr) : BOOLEAN;
```

SFGetFile passes this function the file information it gets by calling the File Manager procedure PBGetFInfo (see the *** forthcoming *** File Manager manual for details). The function selects which files should appear in the dialog by returning FALSE for every file that should be

shown and TRUE for every file that shouldn't be shown.

(note)
As described in the File Manager manual, a flag can be set that tells the Finder not to display a particular file's icon on the desktop; this has no effect on whether SFGetFile will list the file name.

If you want to use the standard SFGetFile dialog box, pass NIL for dlgHook; otherwise, see the information for advanced programmers below.

Like SFPutFile, SFGetFile repeatedly calls the Dialog Manager procedure ModalDialog. When an event involving an enabled dialog item occurs, ModalDialog handles the event and returns the item number, and SFGetFile responds as follows:

- If the Eject or Drive button is clicked, or a disk is inserted, SFGetFile responds as described above under "About the Standard File Package".

- If clicking or dragging occurs in the scroll bar, the contents of the dialog box are redrawn accordingly.

- If a file name is clicked, it's selected and stored in the fName field of the reply record. (SFGetFile keeps track of whether a file name is currently selected, and makes the Open button inactive if not.)

- If the Open button is clicked, SFGetFile returns control to the application with the first field of the reply record set to TRUE.

- If a file name is double-clicked, SFGetFile responds as if the user clicked the file name and then the Open button.

- If the Cancel button in the dialog is clicked, SFGetFile returns control to the application with the first field of the reply record set to FALSE.

If a key (other than a modifier key) is pressed, SFGetFile selects the first file name starting with the character typed. If no file name starts with that character, it selects the first file name starting with a character whose ASCII code is greater than the character typed.

Advanced programmers: You can create your own dialog box rather than use the standard SFGetFile dialog. To do this, you must provide your own dialog template and store it in your application's resource file with the same resource ID that the standard template has in the system resource file:

```
CONST getDlgID = -4000;  {SFGetFile dialog template ID}
```

(note)
The SFPGetFile procedure, described below, lets you use any resource ID for your nonstandard dialog box.

Your dialog template must specify that the dialog window be invisible, and your dialog must contain all the standard items, as listed below. The appearance and location of these items in your dialog may be different. You can make an item "invisible" by giving it a display rectangle that's off the screen. The display rectangle for each in the standard dialog box is given below. The rectangle for the standard dialog box itself is (Ø, Ø, 348, 136).

Item number	Item	Standard display rectangle
1	Open button	(152, 28, 232, 46)
2	Invisible button	(1152, 59, 1232, 77)
3	Cancel button	(152, 9Ø, 232, 1Ø8)
4	UserItem for disk name	(248, 28, 344, 46)
5	Eject button	(256, 59, 336, 77)
6	Drive button	(256, 9Ø, 336, 1Ø8)
7	UserItem for file name list	(12, 11, 125, 125)
8	UserItem for scroll bar	(124, 11, 14Ø, 125)
9	UserItem for gray line	(244, 2Ø, 245, 116)
1Ø	Invisible text (statText)	(1Ø44, 2Ø, 1145, 116)

If your dialog has additional items beyond the the standard ones, or if you want to handle any of the standard items in a nonstandard manner, you must write your own dlgHook function and point to it with dlgHook. Your dlgHook function should have two parameters and return an integer value. For example, this is how it would be declared if it were named MyDlg:

```
FUNCTION MyDlg (item: INTEGER; theDialog: DialogPtr) : INTEGER;
```

Immediately after calling ModalDialog, SFGetFile calls your dlgHook function, passing it the item number returned by ModalDialog and a pointer to the dialog record describing your dialog box. Using these two parameters, your dlgHook function should determine how to handle the event. There are predefined constants for the item numbers of standard enabled items, as follows:

```
CONST getOpen   = 1;  {Open button}
      getCancel = 3;  {Cancel button}
      getEject  = 5;  {Eject button}
      getDrive  = 6;  {Drive button}
      getNmList = 7;  {userItem for file name list}
      getScroll = 8;  {userItem for scroll bar}
```

ModalDialog also returns "fake" item numbers in the following situations, which are detected by its filterProc function:

- When a disk-inserted event occurs, it returns 1ØØ.
- When a key-down event occurs, it returns 1ØØØ plus the ASCII code of the character.

After handling the event (or, perhaps, after ignoring it) your dlgHook function must return an item number to SFGetFile. If the item number is one of those listed above, SFGetFile responds in the standard way;

otherwise, it does nothing.

```
PROCEDURE SFPGetFile (where: Point; prompt: Str255; fileFilter:
            ProcPtr; numTypes: INTEGER; typeList: SFTypeList; dlgHook:
            ProcPtr; VAR reply: SFReply; dlgID: INTEGER; filterProc:
            ProcPtr);
```

SFPGetFile is an alternative to SFGetFile for advanced programmers who want to use a nonstandard dialog box. It's the same as SFGetFile except for the two additional parameters dlgID and filterProc.

DlgID is the resource ID of the dialog template to be used instead of the standard one (so you can use whatever ID you wish rather than the same one as the standard).

The filterProc parameter determines how ModalDialog will filter events when called by SFPGetFile. If filterProc is NIL, ModalDialog does the standard filtering that it does when called by SFGetFile; otherwise, filterProc should point to a function for ModalDialog to execute **after** doing the standard filtering. Note, however, that the standard filtering will detect key-down events only if the dialog template ID is the standard one.

THE DISK INITIALIZATION PACKAGE

The Disk Initialization Package provides routines for initializing disks to be accessed with the Macintosh Operating System's File Manager and Disk Driver. A single routine lets you easily present the standard user interface for initializing and naming a disk; the Standard File Package calls this routine when the user inserts an uninitialized disk. You can also use the Disk Initialization Package to perform each of the three steps of initializing a disk separately if desired.

*** In the Inside Macintosh manual, the documentation of this package will be at the end of the volume that describes the Operating System. ***

You should already be familiar with the following:

- the basic concepts and structures behind QuickDraw, particularly points
- the Toolbox Event Manager
- the File Manager *** the File Manager manual doesn't yet exist ***
- the Package Manager and packages in general

Using the Disk Initialization Package

This section discusses how the routines in the Disk Initialization package fit into the general flow of an application program, and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

The Disk Initialization Package and the resources it uses are automatically read into memory from the system resource file when one of the routines in the package is called. Together, the package and its resources occupy about 2.5K bytes. If the disk containing the system resource file isn't currently in a Macintosh disk drive, the user will be asked to switch disks and so may have to remove the one to be initialized. To avoid this, you can use the DILoad procedure, which explicitly reads the necessary resources into memory and makes them unpurgeable. You would need to call DILoad before explicitly ejecting the system disk or before any situations where it may be switched with another disk (except for situations handled by the Standard File Package, which calls DILoad itself).

(note)

> The resources used by the Disk Initialization Package consist of a single dialog and its associated items, even though the package may present what seem to be a number of different dialogs. A special technique was used to allow the single dialog to contain all possible dialog items with only some of them visible at one time. ***

This technique will be documented in the next draft of the Dialog Manager manual. ***

When you no longer need to have the Disk Initialization Package in memory, call DIUnload. The Standard File Package calls DIUnload before returning.

When a disk-inserted event occurs, the system attempts to mount the volume (by calling the File Manager function PBMountVol) and returns PBMountVol's result code in the high-order word of the event message. In response to such an event, your application can examine the result code in the event message and call DIBadMount if an error occurred (that is, if the volume could not be mounted). If the error is one that can be corrected by initializing the disk, DIBadMount presents the standard user interface for initializing and naming the disk, and then mounts the volume itself. For other errors, it justs ejects the disk; these errors are rare, and may reflect a problem in your program.

(note)

Disk-inserted events during standard file saving and opening are handled by the Standard File Package. You'll call DIBadMount only in other, less common situations (for example, if your program explicitly ejects disks, or if you want to respond to the user's inserting an uninitialized disk when not expected).

Disk initialization consists of three steps, each of which can be performed separately by the functions DIFormat, DIVerify, and DIZero. Normally you won't call these in a standard application, but they may be useful in special utility programs that have a nonstandard interface.

Disk Initialization Package Routines

Assembly-language note: The macros for calling the Disk Initialization Package routines push one of the following routine selectors onto the stack and then invoke _Pack2:

Routine	Selector
DIBadMount	Ø
DIFormat	6
DILoad	2
DIUnload	4
DIVerify	8
DIZero	1Ø

```
PROCEDURE DILoad;
```

DILoad reads the Disk Initialization Package, and its associated dialog and dialog items, from the system resource file into memory and makes them unpurgeable.

(note)

DIFormat, DIVerify, and DIZero don't need the dialog, so if you use only these routines you can call the Resource Manager function GetResource to read just the package resource into memory (and the Memory Manager procedure HNoPurge to make it unpurgeable).

```
PROCEDURE DIUnload;
```

DIUnload makes the Disk Initialization Package (and its associated dialog and dialog items) purgeable.

```
FUNCTION DIBadMount (where: Point; evtMessage: LongInt) : INTEGER;
```

Call DIBadMount when a disk-inserted event occurs if the result code in the high-order word of the associated event message indicates an error (that is, the result code is other than noErr). Given the event message in evtMessage, DIBadMount evaluates the result code and either ejects the disk or lets the user initialize and name it. The low-order word of the event message contains the drive number. The where parameter specifies the location (in global coordinates) of the top left corner of the dialog box displayed by DIBadMount.

If the result code passed is extFSErr, mFulErr, nsDrvErr, paramErr, or volOnLinErr, DIBadMount simply ejects the disk from the drive and returns the result code. If the result code ioErr, badMDBErr, or noMacDskErr is passed, the error can be corrected by initializing the disk; DIBadMount displays a dialog box that describes the problem and asks whether the user wants to initialize the disk. For the result code ioErr, the dialog box shown in Figure D-1 is displayed. (This happens if the disk is brand new.) For badMDBErr and noMacDskErr, DIBadMount displays a similar dialog box in which the description of the problem is "This disk is damaged" and "This is not a Macintosh disk", respectively.

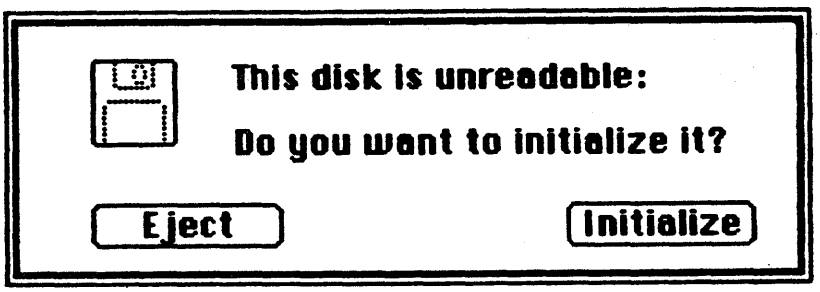


Figure D-1. Disk Initialization Dialog for IOErr

(note)

Before presenting the disk initialization dialog, DIBadMount checks whether the drive contains an already mounted volume; if so, it ejects the disk and returns 2 as its result. This will happen rarely and may reflect an error in your program (for example, you forgot to call DILoad and the user had to switch to the disk containing the system resource file).

If the user responds to the disk initialization dialog by clicking the Eject button, DIBadMount ejects the disk and returns 1 as its result. If the Initialize button is clicked, a box displaying the message "Initializing disk..." appears, and DIBadMount attempts to initialize the disk. If initialization fails, the disk is ejected and the user is informed as shown in Figure D-2; after the user clicks OK, DIBadMount returns a negative result code ranging from firstDskErr to lastDskErr, indicating that a low-level disk error occurred.

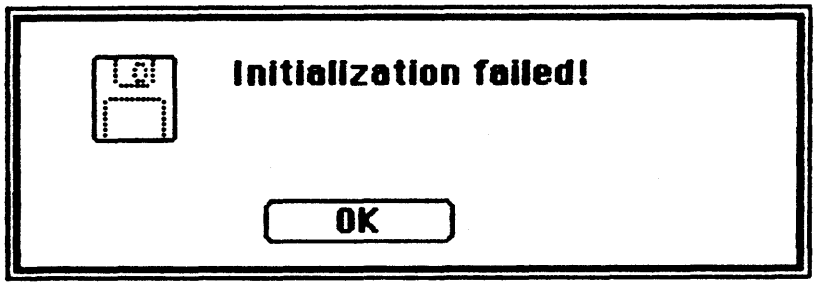


Figure D-2. Initialization Failure Dialog

If the disk is successfully initialized, the dialog box in Figure D-3 appears. After the user names the disk and clicks OK, DIBadMount mounts the volume by calling the File Manager function PBMountVol and returns PBMountVol's result code (noErr if no error occurs).

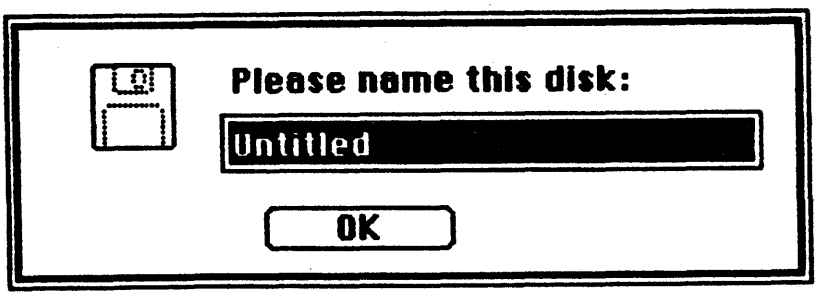


Figure D-3. Dialog for Naming a Disk

Result codes	noErr	No error
	extFSErr	External file system
	mFulErr	Memory full
	nsDrvErr	No such drive
	paramErr	Bad drive number
	volOnLinErr	Volume already on-line
	firstDskErr through lastDskErr	Low-level disk error
Other results	1	User clicked Eject
	2	Mounted volume in drive

```
FUNCTION DIFormat (drvNum: INTEGER) : OSErr;
```

DIFormat formats the disk in the drive specified by the given drive number and returns a result code indicating whether the formatting was completed successfully or failed. Formatting a disk consists of writing special information onto it so that the Disk Driver can read from and write to the disk.

Result codes	noErr	No error
	firstDskErr through lastDskErr	Low-level disk error

```
FUNCTION DIVerify (drvNum: INTEGER) : OSErr;
```

DIVerify verifies the format of the disk in the drive specified by the given drive number; it reads each bit from the disk and returns a result code indicating whether all bits were read successfully or not.

Result codes	noErr	No error
	firstDskErr through lastDskErr	Low-level disk error

```
FUNCTION DIZero (drvNum: INTEGER; volName: Str255) : OSErr;
```

On the unmounted volume in the drive specified by the given drive number, DIZero writes the volume information, a block map, and a file directory as for a volume with no files; the volName parameter specifies the volume name to be included in the volume information. This is the last step in initialization (after formatting and verifying) and makes any files that are already on the volume permanently inaccessible. If the operation fails, DIZero returns a result code indicating that a low-level disk error occurred; otherwise, it mounts the volume by calling the File Manager function PBMountVol and returns PBMountVol's result code (noErr if no error occurs).

Result codes		
	noErr	No error
	badMDBErr	Bad master directory block
	extFSErr	External file system
	ioErr	Disk I/O error
	mFulErr	Memory full
	noMacDskErr	Not a Macintosh volume
	nsDrvErr	No such drive
	paramErr	Bad drive number
	volOnLinErr	Volume already on-line
	firstDskErr through lastDskErr	Low-level disk error

SUMMARY OF THE PACKAGE MANAGER

Constants

```
CONST { Resource IDs for packages }

      dskInit = 2;  {Disk Initialization}
      stdFile = 3;  {Standard File}
      flPoint = 4;  {Floating-Point Arithmetic}
      trFunc  = 5;  {Transcendental Functions}
      intUtil = 6;  {International Utilities}
      bdConv  = 7;  {Binary-Decimal Conversion}
```

Routines

```
PROCEDURE InitPack (packID: INTEGER);
PROCEDURE InitAllPacks;
```

Assembly-Language Information

Constants

```
; Resource IDs for packages

dskInit     .EQU    2  ;Disk Initialization
stdFile     .EQU    3  ;Standard File
flPoint     .EQU    4  ;Floating-Point Arithmetic
trFunc      .EQU    5  ;Transcendental Functions
intUtil     .EQU    6  ;International Utilities
bdConv      .EQU    7  ;Binary-Decimal Conversion
```

SUMMARY OF THE INTERNATIONAL UTILITIES PACKAGE

Constants

```
CONST { Masks for currency format }

      currSymLead   = 16;  {set if currency symbol leads}
      currNegSym    = 32;  {set if minus sign for negative}
      currTrailingZ = 64;  {set if trailing decimal zeroes}
      currLeadingZ  = 128; {set if leading integer zero}

      { Order of short date elements }

      mdy = Ø;  {month day year}
      dmy = 1;  {day month year}
      ymd = 2;  {year month day}

      { Masks for short date format }

      dayLeadingZ = 32;  {set if leading zero for day}
      mntLeadingZ = 64;  {set if leading zero for month}
      century     = 128; {set if century included}

      { Masks for time format }

      secLeadingZ = 32;  {set if leading zero for seconds}
      minLeadingZ = 64;  {set if leading zero for minutes}
      hrLeadingZ  = 128; {set if leading zero for hours}

      { High-order byte of version information }

      verUS      = Ø;
      verFrance  = 1;
      verBritain = 2;
      verGermany = 3;
      verItaly   = 4;
```

Data Types

```
TYPE Intl0Hndl = ^Intl0Ptr;
     Intl0Ptr  = ^Intl0Rec;
```

```
Intl0Rec  = PACKED RECORD
              decimalPt:    CHAR; {decimal point character}
              thousSep:     CHAR; {thousands separator}
              listSep:      CHAR; {list separator}
              currSym1:     CHAR; {currency symbol}
              currSym2:     CHAR;
              currSym3:     CHAR;
              currFmt:      Byte; {currency format}
              dateOrder:    Byte; {order of short date elements}
              shortDateFmt: Byte; {short date format}
              dateSep:      CHAR; {date separator}
              timeCycle:    Byte; {0 if 24-hour cycle, 255 if 12-hour}
              timeFmt:      Byte; {time format}
              mornStr:      PACKED ARRAY[1..4] OF CHAR;
                             {trailing string for first 12-hour cycle}
              eveStr:       PACKED ARRAY[1..4] OF CHAR;
                             {trailing string for last 12-hour cycle}
              timeSep:      CHAR; {time separator}
              time1Suff:    CHAR; {trailing string for 24-hour cycle}
              time2Suff:    CHAR;
              time3Suff:    CHAR;
              time4Suff:    CHAR;
              time5Suff:    CHAR;
              time6Suff:    CHAR;
              time7Suff:    CHAR;
              time8Suff:    CHAR;
              metricSys:    Byte;   {255 if metric, 0 if not}
              intl0Vers:    INTEGER {version information}
            END;

Intl1Hndl = ^Intl1Ptr;
Intl1Ptr  = ^Intl1Rec;
Intl1Rec  = PACKED RECORD
              days:        ARRAY[1..7] OF STRING[15];  {day names}
              months:      ARRAY[1..12] OF STRING[15]; {month names}
              suppressDay: Byte;   {0 for day name, 255 for none}
              longDateFmt: Byte;   {order of long date elements}
              dayleading0: Byte;   {255 for leading 0 in day number}
              abbrLen:     Byte;   {length for abbreviating names}
              st0:         PACKED ARRAY[1..4] OF CHAR; {strings }
              st1:         PACKED ARRAY[1..4] OF CHAR; { for }
              st2:         PACKED ARRAY[1..4] OF CHAR; { long }
              st3:         PACKED ARRAY[1..4] OF CHAR; { date }
              st4:         PACKED ARRAY[1..4] OF CHAR; { format}
              intl1Vers:   INTEGER; {version information}
              localRtn:    INTEGER  {routine for localizing string }
                                    { comparison; actually may be }
                                    { longer than one integer}
            END;

DateForm = (shortDate, longDate, abbrevDate);
```

Routines

```
PROCEDURE IUDateString  (dateTime: LongInt; form: DateForm; VAR result:
                         Str255);
PROCEDURE IUDatePString (dateTime: LongInt; form: DateForm; VAR result:
                         Str255; intlParam: Handle);
PROCEDURE IUTimeString  (dateTime: LongInt; wantSeconds: BOOLEAN; VAR
                         result: Str255);
PROCEDURE IUTimePString (dateTime: LongInt; wantSeconds: BOOLEAN; VAR
                         result: Str255; intlParam: Handle);
FUNCTION  IUMetric :     BOOLEAN;
FUNCTION  IUGetIntl     (theID: INTEGER) : Handle;
PROCEDURE IUSetIntl     (refNum: INTEGER; theID: INTEGER; intlParam:
                         Handle);
FUNCTION  IUCompString   (aStr,bStr: Str255) : INTEGER;  [Pascal only]
FUNCTION  IUMagString    (aPtr,bPtr: Ptr; aLen,bLen: INTEGER) : INTEGER;
FUNCTION  IUEqualString  (aStr,bStr: Str255) : INTEGER;  [Pascal only]
FUNCTION  IUMagIDString  (aPtr,bPtr: Ptr; aLen,bLen: INTEGER) : INTEGER;
```

Assembly-Language Information

Constants

```
; Currency format

currSymLead     .EQU    4   ;set if currency symbol leads
currNegSym      .EQU    5   ;set if minus sign for negative
currTrailingZ   .EQU    6   ;set if trailing decimal zeroes
currLeadingZ    .EQU    7   ;set if leading integer zero

; Order of short date elements

mdy             .EQU    Ø   ;month day year
dmy             .EQU    1   ;day month year
ymd             .EQU    2   ;year month day

; Short date format

dayLeadingZ     .EQU    5   ;set if leading zero for day
mntLeadingZ     .EQU    6   ;set if leading zero for month
century         .EQU    7   ;set if century included

; Time format

secLeadingZ     .EQU    5   ;set if leading zero for seconds
minLeadingZ     .EQU    6   ;set if leading zero for minutes
hrLeadingZ      .EQU    7   ;set if leading zero for hours
```

```
; High-order byte of version information

verUS           .EQU    Ø
verFrance       .EQU    1
verBritain      .EQU    2
verGermany      .EQU    3
verItaly        .EQU    4

; Date form for IUDateString and IUDatePString

shortDate       .EQU    Ø   ;short form of date
longDate        .EQU    1   ;long form of date
abbrevDate      .EQU    2   ;abbreviated long form
```

International Resource Ø Data Structure

decimalPt	Decimal point character
thousSep	Thousands separator
listSep	List separator
currSym	Currency symbol
currFmt	Currency format
dateOrder	Order of short date elements
shortDateFmt	Short date format
dateSep	Date separator
timeCycle	Ø if 24-hour cycle, 255 if 12-hour
timeFmt	Time format
mornStr	Trailing string for first 12-hour cycle
eveStr	Trailing string for last 12-hour cycle
timeSep	Time separator
timeSuff	Trailing string for 24-hour cycle
metricSys	255 if metric, Ø if not
intlØVers	Version information

International Resource 1 Data Structure

days	Day names
months	Month names
suppressDay	Ø for day name, 255 for none
longDateFmt	Order of long date elements
dayleadingØ	255 for leading Ø in day number
abbrLen	Length for abbreviating names
stØ	Strings for long date format
st1	
st2	
st3	
st4	
intl1Vers	Version information
localRtn	Comparison localization routine

Routine Selectors

Routine	Selector
IUDatePString	14
IUDateString	Ø
IUGetIntl	6
IUMagIDString	12
IUMagString	1Ø
IUMetric	4
IUSetIntl	8
IUTimePString	16
IUTimeString	2

SUMMARY OF THE BINARY-DECIMAL CONVERSION PACKAGE

Routines

```
PROCEDURE NumToString (theNum: LongInt; VAR theString: Str255);
PROCEDURE StringToNum (theString: Str255; VAR theNum: LongInt);
```

Assembly-Language Information

Routine Selectors

Routine	Selector
NumToString	Ø
StringToNum	1

SUMMARY OF THE STANDARD FILE PACKAGE

Constants

```
CONST = putDlgID = -3999;  {SFPutFile dialog template ID}

        { Item numbers of enabled items in SFPutFile dialog }

        putSave   = 1;  {Save button}
        putCancel = 2;  {Cancel button}
        putEject  = 5;  {Eject button}
        putDrive  = 6;  {Drive button}
        putName   = 7;  {editText item for file name}

        getDlgID = -4ØØØ;  {SFGetFile dialog template ID}

        { Item numbers of enabled items in SFGetFile dialog }

        getOpen   = 1;  {Open button}
        getCancel = 3;  {Cancel button}
        getEject  = 5;  {Eject button}
        getDrive  = 6;  {Drive button}
        getNmList = 7;  {userItem for file name list}
        getScroll = 8;  {userItem for scroll bar}
```

Data Types

```
TYPE SFReply = RECORD
                 good:    BOOLEAN;    {FALSE if ignore command}
                 copy:    BOOLEAN;    {not used}
                 fType:   OSType;     {file type or not used}
                 vRefNum: INTEGER;    {volume reference number}
                 version: INTEGER;    {file's version number}
                 fName:   STRING[63]  {file name}
               END;

     SFTypeList = ARRAY [Ø..3] OF OSType;
```

Routines

```
PROCEDURE SFPutFile  (where: Point; prompt: Str255; origName: Str255;
                      dlgHook: ProcPtr; VAR reply: SFReply);
PROCEDURE SFPPutFile (where: Point; prompt: Str255; origName: Str255;
                      dlgHook: ProcPtr; VAR reply: SFReply; dlgID:
                      INTEGER; filterProc: ProcPtr);
PROCEDURE SFGetFile  (where: Point; prompt: Str255; fileFilter:
                      ProcPtr; numTypes: INTEGER; typeList: SFTypeList;
                      dlgHook: ProcPtr; VAR reply: SFReply);
```

```
PROCEDURE SFPGetFile (where: Point; prompt: Str255; fileFilter:
                      ProcPtr; numTypes: INTEGER; typeList: SFTypeList;
                      dlgHook: ProcPtr; VAR reply: SFReply; dlgID:
                      INTEGER; filterProc: ProcPtr);
```

DlgHook Function

```
FUNCTION MyDlg (item: INTEGER; theDialog: DialogPtr) : INTEGER;
```

FileFilter Function

```
FUNCTION MyFileFilter (paramBlock: ParmBlkPtr) : BOOLEAN;
```

Standard SFPutFile Items

Item number	Item	Standard display rectangle
1	Save button	(12, 74, 82, 92)
2	Cancel button	(114, 74, 184, 92)
3	Prompt string (statText)	(12, 12, 184, 28)
4	UserItem for disk name	(2Ø9, 16, 295, 34)
5	Eject button	(217, 43, 287, 61)
6	Drive button	(217, 74, 287, 92)
7	EditText item for file name	(14, 34, 182, 5Ø)
8	UserItem for gray line	(2ØØ, 16, 2Ø1, 88)

Resource IDs of SFPutFile Alerts

Alert	Resource ID
Existing file	-3996
Locked disk	-3997
System error	-3995
Disk not found	-3994

Standard SFGetFile Items

Item number	Item	Standard display rectangle
1	Open button	(152, 28, 232, 46)
2	Invisible button	(1152, 59, 1232, 77)
3	Cancel button	(152, 9Ø, 232, 1Ø8)
4	UserItem for disk name	(248, 28, 344, 46)
5	Eject button	(256, 59, 336, 77)
6	Drive button	(256, 9Ø, 336, 1Ø8)
7	UserItem for file name list	(12, 11, 125, 125)
8	UserItem for scroll bar	(124, 11, 14Ø, 125)
9	UserItem for gray line	(244, 2Ø, 245, 116)
1Ø	Invisible text (statText)	(1Ø44, 2Ø, 1145, 116)

Assembly-Language Information

Constants

```
putDlgID        .EQU      -3999  ;SFPutFile dialog template ID

; Item numbers of enabled items in SFPutFile dialog

putSave         .EQU       1     ;Save button
putCancel       .EQU       2     ;Cancel button
putEject        .EQU       5     ;Eject button
putDrive        .EQU       6     ;Drive button
putName         .EQU       7     ;editText item for file name

getDlgID        .EQU      -4000  ;SFGetFile dialog template ID

; Item numbers of enabled items in SFGetFile dialog

getOpen         .EQU       1     ;Open button
getCancel       .EQU       3     ;Cancel button
getEject        .EQU       5     ;Eject button
getDrive        .EQU       6     ;Drive button
getNmList       .EQU       7     ;userItem for file name list
getScroll       .EQU       8     ;userItem for scroll bar
```

Reply Record Data Structure

```
rGood           FALSE if ignore command
rType           File type
rVolume         Volume reference number
rVersion        File's version number
rName           File name
```

Routine Selectors

Routine	Selector
SFGetFile	2
SFPGetFile	4
SFPPutFile	3
SFPutFile	1

SUMMARY OF THE DISK INITIALIZATION PACKAGE

Routines

```
PROCEDURE DILoad;
PROCEDURE DIUnload;
FUNCTION  DIBadMount (where: Point; evtMessage: LongInt) : INTEGER;
FUNCTION  DIFormat   (drvNum: INTEGER) : OsErr;
FUNCTION  DIVerify   (drvNum: INTEGER) : OsErr;
FUNCTION  DIZero     (drvNum: INTEGER; volName: Str255) : OSErr;
```

Assembly-Language Information

Routine Selectors

Routine	Selector
DIBadMount	Ø
DIFormat	6
DILoad	2
DIUnload	4
DIVerify	8
DIZero	1Ø

Result Codes

Name	Value	Meaning
badMDBErr	-6Ø	Bad master directory block
extFSErr	-58	External file system
firstDskErr	-84	First of the range of low-level disk errors
ioErr	-36	Disk I/O error
lastDskErr	-64	Last of the range of low-level disk errors
mFulErr	-41	Memory full
noErr	Ø	No error
noMacDskErr	-57	Not a Macintosh disk
nsDrvErr	-56	No such drive
paramErr	-5Ø	Bad drive number
volOnLinErr	-55	Volume already on-line

GLOSSARY

ligature: A character that combines two letters.

list separator: The character that separates numbers, as when a list of numbers is entered by the user.

package: A set of data structures and routines that's stored as a resource and brought into memory only when needed.

routine selector: An integer that's pushed onto the stack before the _PackN macro is invoked, to identify which routine to execute. (N is the resource ID of a package; all macros for calling routines in the package expand to invoke _PackN.)

thousands separator: The character that separates every three digits to the left of the decimal point.

COMMENTS?

Macintosh User Education encourages your comments on this manual.

- What do you like or dislike about it?

- Were you able to find the information you needed?

- Was it complete and accurate?

- Do you have any suggestions for improvement?

Please send your comments to the author (indicated on the cover page) at 10460 Bandley Drive M/S 3-G, Cupertino CA 95014. Mark up a copy of the manual or note your remarks separately. (We'll return your marked-up copy if you like.)

Thanks for your help!

MACINTOSH USER EDUCATION

Printing From Macintosh Applications /PRINTING/PRINT

See Also: The Resource Manager: A Programmer's Guide
QuickDraw: A Programmer's Guide
The Font Manager: A Programmer's Guide
The Dialog Manager: A Programmer's Guide
The Structure of a Macintosh Application
Programming Macintosh Applications in Assembly Language

Modification History: First Draft S. Chernicoff & B. Hacker 6/11/84

ABSTRACT

Macintosh applications can print information on any variety of printer the user has connected to the Macintosh by calling Printing Manager routines. Advanced programmers can also call the Printer Driver to implement alternate, low-level printing techniques. This manual describes the Printing Manager and Printer Driver.

TABLE OF CONTENTS

ABOUT THIS MANUAL

Macintosh applications can print information on any variety of printer the user has connected to the Macintosh by calling the Printing Manager routines in the User Interface Toolbox. Advanced programmers can also call the Printer Driver to implement alternate, low-level printing techniques. This manual describes the Printing Manager and Printer Driver. *** It will eventually become part of the comprehensive Inside Macintosh manual. ***

Like all Toolbox documentation, this manual assumes you're familiar with the Macintosh User Interface Guidelines, Lisa Pascal, and the Macintosh Operating System's Memory Manager. You should also be familiar with the following:

- resources, as described in the Resource Manager manual
- the use of QuickDraw, as described in the QuickDraw manual, particularly bit images, rectangles, bitMaps, and pictures
- the use of fonts, as described in the Font Manager manual
- the basic concepts of dialogs, as described in the Dialog Manager manual
- files and volumes, as described in the File Manager manual
- device drivers, as described in the Device Manager manual, *** doesn't yet exist *** if you're interested in writing your own Printer Driver

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly-language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it.

The manual begins with an overview of the Printing Manager and what you can do with it. It then discusses the basics about printing: the various methods of printing available; the relationship between printing and the Finder; and the Printing Manager's use of dialogs and data structures, the most important of which is the print record.

Next, a section on using the Printing Manager introduces its routines and tells how they fit into the flow of your application. This is followed by detailed descriptions of all Printing Manager procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that won't interest all readers. Special information is given about the Printer Driver and the format of resource files used when printing, for programmers interested in writing their own Printer Driver.

Finally, there's a summary of the Printing Manager for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE PRINTING MANAGER

The Printing Manager is the part of the Macintosh User Interface Toolbox that's used to print text or graphics on a printer. It's not contained in the Macintosh ROM; it must be read from a resource file before it can be used. The Printing Manager provides your application with:

- two standard printing methods, and the ability to define two more
- a standard dialog for the user to specify the paper size and page orientation they're using, so you can easily implement a Page Setup command in your File menu
- a standard dialog for the user to specify the method of printing, which pages to print, and so on, so you can easily implement a Print command in your File menu
- the ability to perform background processing while the Printing Manager is printing
- a way to abort printing when the user types Command-period

The Printing Manager is designed such that an application need never be concerned with what kind of printer the user has connected to the Macintosh; an application uses the same routine calls to print with all varieties of printers.

This printer independence is possible because the Printing Manager uses separate, printer-specific code to implement its routines for each different variety of printer. While the code for some Printing Manager routines (such as those that begin and end printing sessions), is contained wholly within the Printing Manager itself, the code for other routines (such as those that do the actual printing) depends on the printer being used and is contained in a separate printer resource file on the user's disk. The Printing Manager dispatches calls to these routines, first loading the code into memory if necessary.

Although the actual routines of the Printing Manager differ for each variety of printer, your application uses the same Printing Manager calls to print on all varieties of printers. The user "installs" a new printer by giving the Printing Manager a new printer resource file to work with (Figure 1). Printer installation is transparent to you application, and you needn't be concerned with it.

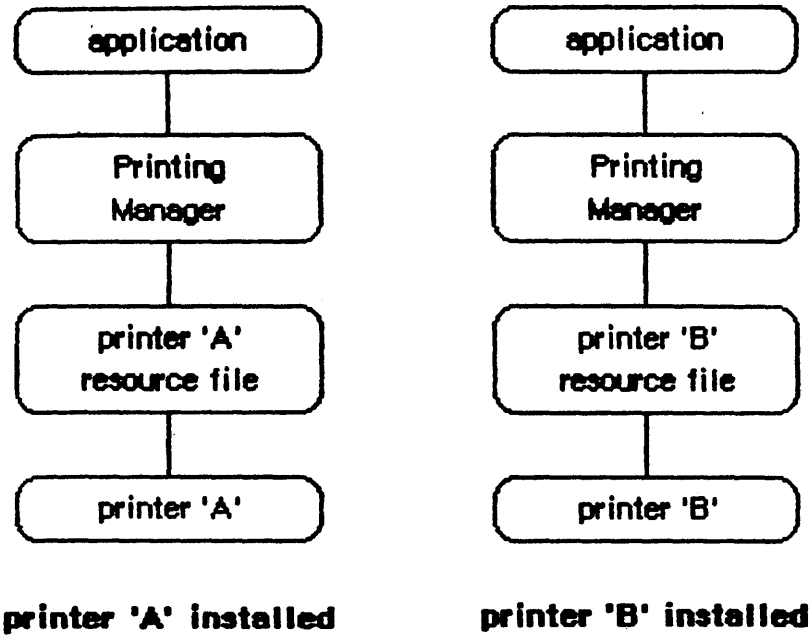


Figure 1. Printer Installation

Each printer resource file also contains a device driver that communicates between the Printing Manager and the printer. Because the actual routines of the device driver differ for each variety of printer, there exists a different device driver for each printer. The Printing Manager routines used to call a printer's device driver are the same, regardless of printer variety; this manual will refer to the device driver of the currently installed printer as the Printer Driver.

You define the image to be printed by using a printing port, a special QuickDraw grafPort customized for printing:

```
TYPE TPPrPort = ^TPrPort;
     TPrPort  = RECORD
                  gPort:  GrafPort; {grafPort to draw in}
                  gProcs: QDProcs;  {pointers to drawing routines}
                  {more fields for internal use only}
                END;
```

The Printing Manager gives you a printing port when you prepare to print a document. You print text and graphics by drawing into this port with QuickDraw, just as if you were drawing on the screen. The Printing Manager installs its own versions of QuickDraw's low-level drawing routines in the printing port, causing your higher-level QuickDraw calls to drive the printer instead of drawing on the screen. GProcs contains pointers to these low-level drawing routines.

(note)
To convert a pointer to a printing port into an equivalent grafPtr for use with QuickDraw, you can use the following variant record type:

```
TYPE TPPort = PACKED RECORD
                CASE INTEGER OF
                  Ø: (pGPort:  GrafPtr);
                  1: (pPrPort: TPPrPort)
              END;
```

METHODS OF PRINTING

The Printing Manager supports two different methods of printing documents: draft and spool. In draft printing, your QuickDraw calls are converted directly into command codes the printer understands, which are then immediately used to drive the printer. Each element of the image is printed as soon as you request it; as you move around to various coordinates within the grafPort, the print head moves to the corresponding positions on the printed page. Draft printing uses the printer's native font and graphics capabilities and probably won't produce an image matching the one on the screen. This method of printing is more direct than spool printing, but it can also be cumbersome, especially for graphics. Draft printing is most appropriate for making quick copies of text documents, which are printed straight down the page from top to bottom and left to right. Depending on the printer and what you're printing, draft printing may not even be possible; for instance, not all printers are capable of moving the paper backwards (toward the top of the page).

Spooling and spool printing are complementary halves of a two-stage process. First you cause the Printing Manager to write out (spool) a representation of your document's printed image to a disk file. This spool file is later read back in, each page is imaged (converted into an array of dots at the appropriate resolution), and the result is sent to the printer in a single pass from top to bottom. Spool printing uses QuickDraw and the Font Manager's graphics and font capabilities to produce an image closely matching the one on the screen.

(note)

> The internal format of spool files is private to the Printing Manager and may vary from one printer to another. This means that spool files destined for one printer can't necessarily be printed on another. In spool files for the Imagewriter printer, each page is stored in the form of a QuickDraw picture. It's envisioned that most other printers will use this same approach, but there may be exceptions.

Spooling and spool printing are two separate stages because spool printing a document takes a lot of space--typically from 2ØK to 4ØK for the printing code, buffers, and fonts, but spooling a document takes only about 3K. When spooling a document, large portions of your application's code and data may be needed in memory; when spool printing, most of your application's code and data are no longer

needed. Normally you'll make your printing code a separate program segment, so you can swap the rest of your code and data out of memory during printing and swap it back in after you're finished.

If your application can't afford the space required by spool printing, it can just perform the spooling stage, and leave the spool file on the disk for the user to print later from the Finder (see next section). The maximum number of pages in a spool file is defined by the following constant *** it may increase *** :

```
CONST iPFMaxPgs = 128; {maximum number of pages in a spool file}
```

(note)

Advanced programmers: In addition to draft printing and spooling, you can define as many as two more of your own methods of document printing for any given printer. (No such additional printing methods are currently defined for the Imagewriter.) There are also a number of low-level printing methods available, such as bitmap printing, text streaming, and screen printing. These methods are discussed in the section "Using a Printer Driver".

Imaging During Spool Printing

The bit image for a typical page is too big to fit in memory all at once. For instance, at the highest resolution of the Imagewriter printer (160 dots per inch horizontally by 144 vertically), an 8-by-10 1/2-inch page image contains approximately a quarter megabyte of information, or twice the total memory capacity of the Macintosh. So instead of imaging and printing the entire page at once, the page has to be broken into bands small enough to fit in memory. During spool printing the Printing Manager actually images each band individually, adjusting the fields of the printing port to limit the actual drawing to the boundaries of the band. It then prints the resulting bit image before imaging the next band. A page can be broken into bands ("scanned") in any of four ways. Figure 2 shows the four possible scan directions of a printing port.

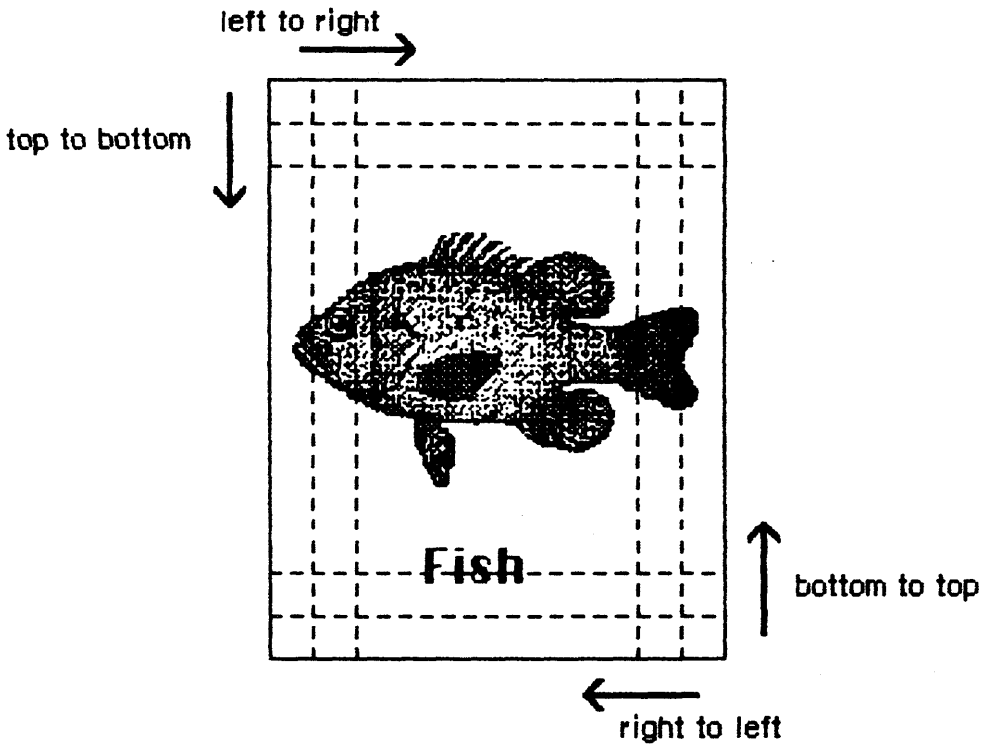


Figure 2. Scan Directions

The bands are always printed from top to bottom relative to the physical sheet of paper; the scan direction determines the correspondence between these printed bands and the dots of the image. If the long dimension of the paper runs vertically with respect to the image, the page is said to be in portrait orientation; if the long dimension runs horizontally, the page is in landscape orientation. In practice, portrait pages are normally scanned from top to bottom and landscape pages from left to right.

PRINTING FROM THE FINDER

The Macintosh user can choose to print from the Finder as well as from an application. Your application should support both alternatives.

To print a document from the Finder, the user selects the document's icon and chooses the Print command from the File menu. When the Print command is chosen, the Finder starts up the document's application, and passes information to the application indicating that the file is to be printed rather than opened. The application is then expected to print the document, preferably without doing its entire startup sequence. It may choose to do any of the following:

- Draft-print the document.
- Spool the document to a file and then print it immediately.
- Spool the document to a file and leave it for the user to print later via the Printer program (descibed below).

If your application writes spool files on a disk and then doesn't spool print them, it's up to the user to print them. The user simply selects the spool file's icon (Figure 3) and chooses the Print command from the File menu. When the Print command is chosen, the Finder starts up a special program called Printer, which spool prints spool files. It's provided as a utility for use with programs that don't do their own spool printing. Its main purpose is to read a spool file, image it, and print it.

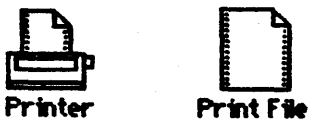


Figure 3. Icons for the Printer Program and Spool Files

Spool files can be identified by their file type and creator:

```
CONST lPfType = $5Ø46484C; {spool file type 'PFIL'}
      lPfSig  = $5Ø535953; {spool file creator 'PSYS'}
```

(note)
The details of the Finder interface are discussed in The Structure of a Macintosh Application.

*** This method of spool printing may be temporary. Currently, the easiest way for your application to do printing is to leave spool files on the disk and rely on the user to print them via Printer. Eventually Printer may be eliminated and one of the following solutions will be employed: The process will remain the same, and the code of Printer will be integrated into the Finder; or your application will be required to do spool printing itself. ***

PRINT RECORDS AND DIALOGS

For every printing operation, your application needs to determine the following:

- the resolution and other characteristics of the printer being used
- the dimensions of the printed image and of the physical sheet of paper
- the printing method to be used (draft or spool)
- the name of the spool file, if applicable
- which pages of the document to print
- how many copies to print
- an optional background procedure to be run during idle times in the printing process (discussed later)

This information is contained in a data structure called a print record. The Printing Manager fills in most of the print record for you. Some values depend on the variety of printer installed in the Printing Manager; others are set as a result of dialogs with the user.

(note)
Whenever you save a document, it's recommended that you write an appropriate print record in the document's file (see the "Printing Resources" section). This allows the document to "remember" its own printing parameters for use the next time it's printed.

(note)
If you try to use a print record that's invalid for the current version of the Printing Manager or for the printer installed in the Printing Manager, the Printing Manager will correct the record by filling it with default values.

The information in the print record that can vary from one printing job to the next is obtained from the user by means of dialogs. The Printing Manager uses two standard dialogs for this purpose. The style dialog includes the paper size and page orientation (Figure 4). This dialog is conventionally associated with a Page Setup command in the application.

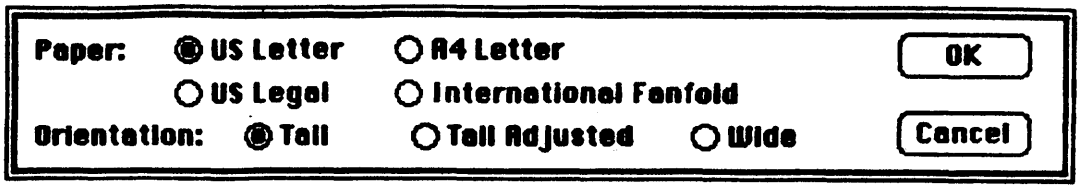


Figure 4. The Standard Style Dialog

The job dialog, normally associated with the application's Print command, requests information on how to print the document **this time**, such as the method of printing (draft or spool), the print quality (for printers that offer a choice of resolutions), the type of paper feed (such as fanfold or cut-sheet), the range of pages to be printed, and the number of copies (Figure 5).

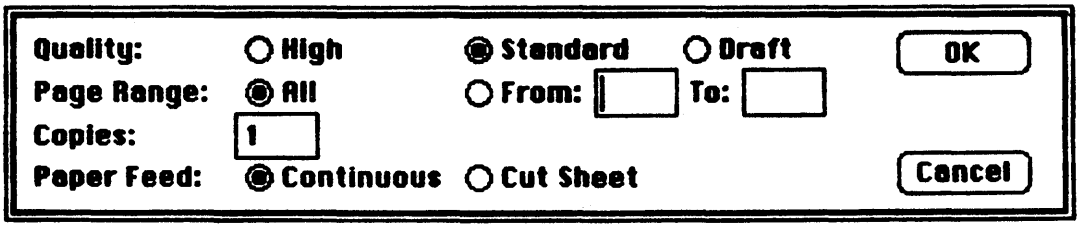


Figure 5. The Standard Job Dialog

Print records are referred to by handles. Their structure is as follows:

```
TYPE THPrint = ^TPPrint;
     TPPrint = ^TPrint;
     TPrint  = RECORD
                 iPrVersion: INTEGER;  {Printing Manager version}
                 prInfo:     TPrInfo;  {printer information}
                 rPaper:     Rect;     {paper rectangle}
                 prStl:      TPrStl;   {style information}
                 prInfoPT:   TPrInfo;  {copy of prInfo}
                 prXInfo:    TPrXInfo; {band information}
                 prJob:      TPrJob;   {job information}
                 printX:     ARRAY [1..19] OF INTEGER
                                       {used internally}
               END;
```

IPrVersion identifies the version of the Printing Manager that initialized this print record.

Most of the other fields of the print record are "subrecords" containing various parts of the overall printing information; these are discussed in separate sections below.

Assembly-language note: The global constant iPrintSize equals the length in bytes of a print record.

The Printer Information Subrecord

The printer information subrecord (field prInfo of the print record) describes the characteristics of the particular printer you're using. Its contents are set by the Printing Manager when it initializes the print record. All applications will need to refer to the information it contains. (The prInfoPT field of the print record is a copy of the prInfo field and is used internally by the Printing Manager during printing.)

The printer information subrecord is defined as follows:

```
TYPE TPrInfo = RECORD
                 iDev:  INTEGER; {driver information}
                 iVRes: INTEGER; {printer vertical resolution}
                 iHRes: INTEGER; {printer horizontal resolution}
                 rPage: Rect     {page rectangle}
               END;
```

The iDev field contains information used by QuickDraw and the Font Manager for selecting fonts for the printer. The high-order byte is the reference number of the Printer Driver, -3. The low-order byte contains device-specific information on how the printer is being used. For example, for the Imagewriter printer, bit Ø specifies high (1) or low (Ø) resolution and bit 1 specifies portrait (1) or landscape (Ø) orientation.

(note)
If you store this word into the device field of a grafPort, you can use the QuickDraw routines CharWidth, StringWidth, TextWidth, and GetFontInfo to ask for information about a font drawn on that device.

IVRes and iHRes give the vertical and horizontal resolution of the printer, in dots per inch.

RPage is the page rectangle, representing the boundaries of the printable page. Its top left corner always has coordinates (Ø,Ø); the coordinates of the bottom right corner give the maximum page height and width attainable on the given printer, in dots. Typically these are slightly less than the physical dimensions of the paper, because of the printer's mechanical limitations.

The results of the style dialog conducted with the user determine the values of the iVRes, iHRes, and rPage fields. For example, with the

Imagewriter printer, the style dialog's three orientation buttons yield the following:

Button	Orientation	IVRes	IHRes
Tall	Portrait	8Ø	72
Tall adjusted	Portrait	72	72
Wide	Landscape	72	72

The physical paper size is given by the rPaper field of the print record. This paper rectangle is outside of the page rectangle: it defines the physical boundaries of the paper in the same coordinate system as rPage (see Figure 6). Thus the top left coordinates of the paper rectangle are typically negative and its bottom right coordinates are greater than those of the page rectangle.

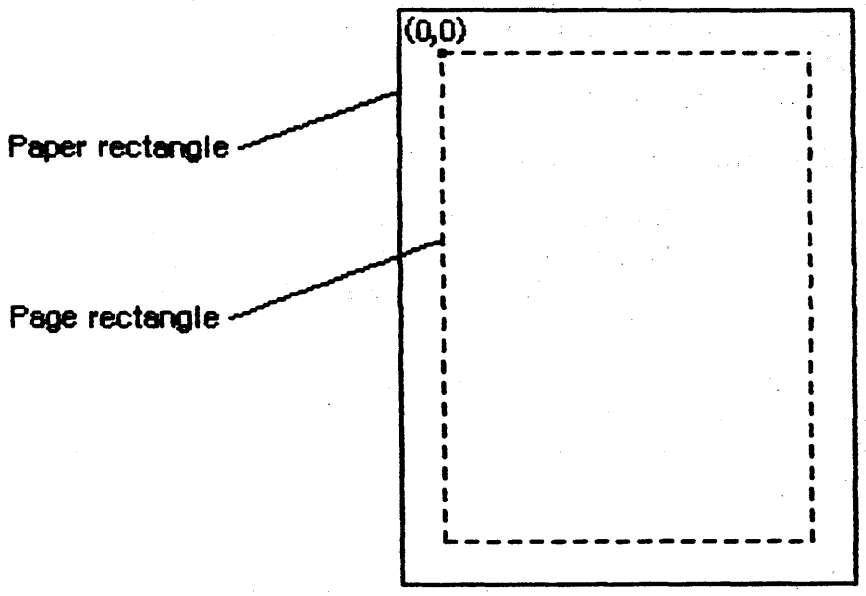


Figure 6. Page and Paper Rectangles

The Style Subrecord

The style subrecord (field prStl of the print record) describes the type and size of paper used in the printer. The contents of the style subrecord are normally set by the Printing Manager after dialogs with the user, and only advanced programmers need be concerned with them.

The style subrecord is defined as follows:

```
TYPE TPrStl = RECORD
                wDev:   TWord;      {used internally}
                iPageV: INTEGER;    {paper height}
                iPageH: INTEGER;    {paper width}
                bPort:  SignedByte; {printer or modem port}
                feed:   TFeed       {paper type}
              END;
```

IPageV and iPageH give the physical dimensions of the paper, in 12Øths of an inch. The user can set them by choosing a standard paper size (such as U.S. Letter, U.S. Legal, or European A4) from the style dialog. The number of units per inch is defined by the following constant:

```
CONST iPrPgFract = 12Ø; {units per inch of paper dimension}
```

BPort designates which port on the back of the Macintosh the printer is connected to: Ø for the printer port, 1 for the modem port. *** Currently the Printing Manager ignores this value, and instead uses the global variable sPPrint. ***

Feed identifies the type of paper feed being used:

```
TYPE TFeed = (feedCut,     {hand-fed, individually cut sheets}
              feedFanfold, {continuous-feed fanfold paper}
              feedMechCut, {mechanically fed cut sheets}
              feedOther);  {other types of paper}
```

The user sets this field by choosing Continuous or Cut Sheet from the job dialog. When Cut Sheet is chosen, the printer will pause at the end of each page and a dialog box will prompt the user to insert the next sheet.

The Job Subrecord

The job subrecord (field prJob of the print record) contains information about a particular printing job. Its contents are normally set by the Printing Manager as a result of a job dialog with the user.

The job subrecord is defined as follows:

```
TYPE TPrJob = RECORD
                iFstPage:  INTEGER;    {first page to print}
                iLstPage:  INTEGER;    {last page to print}
                iCopies:   INTEGER;    {number of copies}
                bJDocLoop: SignedByte; {printing method}
                fFromUsr:  BOOLEAN;    {TRUE if called from application}
                pIdleProc: ProcPtr;    {background procedure}
                pFileName: TPStr8Ø;    {spool file name}
                iFileVol:  INTEGER;    {volume reference number}
                bFileVers: SignedByte; {version number of spool file}
                bJobX:     SignedByte  {not used}
              END;
```

```
TPStr8Ø = ^TStr8Ø;
TStr8Ø  = STRING[8Ø];
```

Most programmers need only be concerned with the bJDocLoop, pFileName, and pIdleProc fields. BJDocLoop represents the method of printing to use. The user sets this field by choosing High, Standard, or Draft from the job dialog. BJDocLoop should be one of the following predefined constants:

```
CONST bDraftLoop = Ø; {draft printing}
      bSpoolLoop = 1; {spooling}
      bUser1Loop = 2; {printer-specific, method 1}
      bUser2Loop = 3; {printer-specific, method 2}
```

If you're spool printing, it's a good idea to give each file you spool to the disk a different name, in the pFileName field, so that it doesn't overwrite any other spool files on the disk. PFileName is initialized to NIL, denoting the default file name found in the printer resource file. *** (Currently the default file name is 'Print File'.) ***

IFstPage and iLstPage designate the first and last pages to be printed. The Printing Manager knows nothing about any page numbering placed by an application within a document, and always considers the first printable page to be page 1. For example, if iFstPage is 2, the Printing Manager will print the second page in the document, regardless of how the page is actually numbered. If you're draft printing, you'll need to use the value of iCopies to determine the number of copies to print (the Printing Manager automatically handles multiple copies for spooling).

FFromUsr is TRUE when the Printing Manager is called from an application program, FALSE when it's called from the Printer program. PIdleProc is a pointer to the background procedure (explained below) for this printing operation. In a newly initialized print record this field is set to NIL, designating the default background procedure. This procedure just polls the keyboard and cancels further printing if the user types Command-period. You can install a background procedure of your own by storing directly into the pIdleProc field.

For spooling operations, iFileVol and bFileVers are the volume reference number and version number of the spool file. IFileVol and bFileVers are both initialized to Ø. You can override the default settings by storing directly into these fields.

The Band Information Subrecord

The band information subrecord (field prXInfo of the print record) contains information about the way a page will be imaged during spool printing. Its contents are set by the Printing Manager, and most programmers needn't be concerned with it.

The band information subrecord is defined as follows:

```
TYPE TPrXInfo = RECORD
                  iRowBytes: INTEGER;     {bytes per row}
                  iBandV:    INTEGER;     {vertical dots}
                  iBandH:    INTEGER;     {horizontal dots}
                  iDevBytes: INTEGER;     {size of bit image}
                  iBands:    INTEGER;     {bands per page}
                  bPatScale: SignedByte;  {used by QuickDraw}
                  bUlThick:  SignedByte;  {underline thickness}
                  bUlOffset: SignedByte;  {underline offset}
                  bUlShadow: SignedByte;  {underline descender}
                  scan:      TScan;       {scan direction}
                  bXInfoX:   SignedByte   {not used}
                END;
```

IRowBytes is the number of bytes in each row of the band's bit image, iBandV and iVBandH are the dimensions of the band in dots, iDevBytes is the number of bytes of memory needed to hold the bit image, and iBands is the number of bands per page.

BPatScale is used by QuickDraw when it scales patterns to the resolution of the printer. BUlThick, bUlOffset, and bUlShadow are used for underlining text; they stand for the thickness of the underline, its offset below the base line, and the width of the break around descenders, all in dots. The scan field specifies the scan direction for banding as a value of type TScan:

```
TYPE TScan = (scanTB,  {scan top to bottom}
              scanBT,  {scan bottom to top}
              scanLR,  {scan bottom to top}
              scanRL); {scan right to left}
```

BACKGROUND PROCESSING

As mentioned above, the job subrecord includes a pointer, pIdleProc, to an optional background procedure to be run whenever the Printing Manager has directed output to the printer and is waiting for the printer to finish. The background procedure takes no parameters and returns no result; the Printing Manager simply runs it at every opportunity. There's no limit to the length of time that a background procedure can execute, but beyond a certain length of time printing will be slowed.

If you don't designate a background procedure, the Printing Manager will use one by default that just polls the keyboard and cancels further printing if the user types Command-period. In this case you should display an alert box to inform the user that the Command-period option is available. It's suggested, however, that instead of relying on this method, you supply your own background procedure to give the user a more convenient way to cancel printing. For instance, you might put up a dialog box with a Cancel button the user can click with the mouse; or, in a background procedure that runs your application, you might replace the Print command with Stop Print.

While printing from a spool file, the Printing Manager maintains a printer status record in which it reports on the progress of the printing operation:

```
TYPE TPrStatus = RECORD
                   iTotPages:  INTEGER;  {total number of pages}
                   iCurPage:   INTEGER;  {page being printed}
                   iTotCopies: INTEGER;  {number of copies}
                   iCurCopy:   INTEGER;  {copy being printed}
                   iTotBands:  INTEGER;  {bands per page}
                   iCurBand:   INTEGER;  {band being printed}
                   fPgDirty:   BOOLEAN;  {TRUE if started printing page}
                   fImaging:   BOOLEAN;  {TRUE if imaging}
                   hPrint:     THPrint;  {print record}
                   pPrPort:    TPPrPort; {printing port}
                   hPic:       PicHandle {used internally}
                 END;
```

FPgDirty is TRUE if anything has been printed yet on the current page, FALSE if not; fImaging is TRUE while a band is being imaged, FALSE while it's being printed. HPrint is a handle to the print record for this printing operation; pPrPort is a pointer to the printing port.

Your background procedure can use this information--for example, to display a progress report on the screen ("Now printing copy 3 of 5, page 7 of 12").

(note)

The Printing Manager only calls your background procedure while it's printing. If you want your background procedure to execute during spooling, you'll have to call it yourself.

Advanced programmers can use background processing in a variety of useful ways. For example, with a background procedure that performs one pass through your main program loop, you can achieve the effect of concurrent printing. That is, your application can continue to run while the printing is taking place, although there may be some degradation in performance. The user is given the illusion that the printing is going on "in the background" behind the application. (In reality, of course, it's the application that's running in the background behind the printing task.)

(warning)

You have to be careful in the way you write your background procedure, to avoid a number of subtle concurrency problems that may arise. For instance, if the background procedure uses QuickDraw, it must be sure to restore the printing port as the current port before returning. It's particularly important not to attempt any printing from within the background procedure: the Printing Manager is **not** reentrant! If you use a background procedure that runs your application concurrently with printing, it should disable all menu items having to do with printing, such as Page Setup and Print.

USING THE PRINTING MANAGER

This section discusses how the Printing Manager routines fit into the general flow of your program and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

To use the Printing Manager, you must have previously initialized QuickDraw, the Font Manager, the Window Manager, the Menu Manager, TextEdit, and the Dialog Manager. The first Printing Manager routine to call is PrOpen, which opens the printer resource file. The last routine to call is PrClose, which closes the Printer Driver and the printer resource file.

(note)

PrOpen and PrClose are meant to be called once each, at the beginning and end of your application. However, if space is particularly critical, you may prefer to bracket every Printing Manager call with a PrOpen and a PrClose. This frees the space occupied by various Printing Manager data structures when they're not in use.

Before printing a document, you need a properly filled out print record. You can either use an existing print record (for instance, from a document) or initialize one to the current default settings by calling PrintDefault. If you use an existing print record, you should call PrValidate to make sure it's valid for the current version of the Printing Manager and for the currently installed printer.

When the user chooses the Page Setup commmand, call PrStlDialog to ask about the paper size and page orientation. From the printer information subrecord you can then determine where each page break occurs.

When the user chooses the Print commmand, call PrJobDialog to ask the user for specific information about that printing job. To apply the results of one job dialog to several documents (when printing from the Finder, for example), call PrJobMerge.

To draft print or spool a document, begin by calling PrOpenDoc, which returns a printing port customized for draft printing or spooling (depending on the bJDocLoop field of the job subrecord). You can then print or spool your document by "drawing" into this printing port with QuickDraw, using the values in the printer information subrecord to adjust for the parameters of the printer. Call PrOpenPage and PrClosePage at the beginning and end of each page, and PrCloseDoc at the end of the entire document. Each page is either printed immediately (draft printing) or written to the disk as part of a spool file (spooling).

To print a spool file, swap as much of your program out of memory as you can, and then call PrPicFile.

Call PrError to check for errors caused by a Printing Manager routine. To cancel a printing operation in progress, use PrSetError. Be sure to call PrCloseDoc or PrClosePage after you cancel printing in progress.

PRINTING MANAGER ROUTINES

This section describes the procedures and functions that make up the Printing Manager. They're presented in their Pascal form; for information on using them from assembly language, see Programming Macintosh Applications in Assembly Language.

Initialization and Termination

PROCEDURE PrOpen;

PrOpen prepares the Printing Manager for use. It opens the Printer Driver and the printer resource file. If either of these items is missing, or if the printer resource file is not properly formed, PrOpen will do nothing, and PrError will return a Resource Manager result code.

PROCEDURE PrClose;

PrClose releases the memory used by the Printing Manager. It closes the printer resource file, allowing the file's resource map to be removed from memory. It *** currently *** doesn't close the Printer Driver, however, since the driver may have been opened before the PrOpen call was issued.

Print Records and Dialogs

```
PROCEDURE PrintDefault (hPrint: THPrint);
```

PrintDefault fills the fields of a print record with the current default values stored in the printer resource file. HPrint is a handle to the record, which may be a new print record that you've just allocated or an existing one (from a document, for example).

```
FUNCTION PrValidate (hPrint: THPrint) : BOOLEAN;
```

PrValidate checks the contents of a print record for compatibility with the current version of the Printing Manager and with the installed printer. If the record is valid, the function returns FALSE (no change); if invalid, the record is adjusted to the current default values, taken from the printer resource file, and the function returns TRUE.

PrValidate also updates the print record to reflect the current settings in the style and job subrecords. These changes have no effect on the function's Boolean result.

```
FUNCTION PrStlDialog (hPrint: THPrint) : BOOLEAN;
```

PrStlDialog conducts a style dialog with the user to determine the paper size and paper orientation being used. The initial settings displayed in the dialog box are taken from the current values in the print record. If the user confirms the dialog, the results of the dialog are saved in the print record and the function returns TRUE; otherwise the print record is left unchanged and the function returns FALSE.

(note)

> If the print record was taken from a document, you should update its contents in the document's file if PrStlDialog returns TRUE. This makes the results of the style dialog "stick" to the document.

```
FUNCTION PrJobDialog (hPrint: THPrint) : BOOLEAN;
```

PrJobDialog conducts a job dialog with the user to determine the printing quality, number of pages to print, and so on. The initial settings displayed in the dialog box are taken from the current values in the print record. If the user confirms the dialog, both the print record and the printer resource file are updated (so that the user's choices "stick" to the printer) and the function returns TRUE; otherwise the print record and printer resource file are left unchanged and the function returns FALSE.

(note)

If the job dialog is associated with your application's Print command, you should proceed with the requested printing operation if PrJobDialog returns TRUE. If the print record was taken from a document, you should update its contents in the document's file.

```
PROCEDURE PrJobMerge (hPrintSrc,hPrintDst: THPrint);
```

PrJobMerge copies the job subrecord from one print record (hPrintSrc) to another (hPrintDst) and updates the destination record's printer information, band information, and paper rectangle, based on information in the job subrecord. This allows the information in the job subrecord to be used for a group of related jobs.

Draft Printing and Spooling

```
FUNCTION PrOpenDoc (hPrint: THPrint; pPrPort: TPPrPort; pIOBuf: Ptr)
          : TPPrPort;
```

PrOpenDoc initializes a printing port for use in printing a document, makes it the current port, and returns a pointer to it. HPrint is a handle to the print record for this printing operation. The printing port is customized for draft printing or spooling, depending on the setting of the bJDocLoop field in the job subrecord. For spooling, the spool file's name, volume reference number, and version number are taken from the job subrecord.

PPrPort is a pointer to the storage to be used for the printing port. If this parameter is NIL, PrOpenDoc will allocate a new printing port for you. Similarly, pIOBuf points to an area of memory to be used as an input/output buffer; if it's NIL, PrOpenDoc will use the volume buffer for the spool file's volume.

(note)

The pPrPort and pIOBuf parameters are provided because both the printing port and the input/output buffer are nonrelocatable objects. To avoid cluttering the heap with such objects, you have the opportunity to allocate them yourself and pass them to PrOpenDoc. Most of the time you'll just set both of these parameters to NIL.

(note)

Newly created printing ports use the system font (since they're grafPorts), but newly created windows use the application font. Be sure the font you use in the printing port is the same as the font in your application window if you want the text in both places to match.

```
PROCEDURE PrOpenPage (pPrPort: TPPrPort; pPageFrame: TPRect);
```

PrOpenPage begins a new page in the document associated with the given printing port. The page is printed only if it falls within the page range designated in the job subrecord.

For spooling, the pPageFrame parameter points to a rectangle that will be used as the QuickDraw picture frame for this page:

```
TYPE TPRect = ^Rect;
```

When the spool file is later printed, this rectangle will be scaled (via the QuickDraw DrawPicture procedure) to coincide with the page rectangle in the printer information subrecord. Unless you want the printout to be scaled, you should set pPageFrame to NIL--this uses the current page rectangle as the picture frame, and the page will be printed with no scaling.

```
PROCEDURE PrClosePage (pPrPort: TPPrPort);
```

PrClosePage finishes up the current page of the document associated with the given printing port. For draft printing, it ejects the page from the printer and, if necessary, alerts the user to insert another; for spooling, it closes the picture representing the current page.

```
PROCEDURE PrCloseDoc (pPrPort: TPPrPort);
```

PrCloseDoc finishes up the printing of the document associated with the given printing port. For draft printing, it issues a form feed and a reset command to the printer; for spooling, it closes the file if the spooling was successfully completed or deletes it the file if the spooling was unsuccessful.

Spool Printing

```
PROCEDURE PrPicFile (hPrint: THPrint; pPrPort: TPPrPort; pIOBuf: Ptr;
          pDevBuf: Ptr; VAR prStatus: TPrStatus);
```

PrPicFile images and prints a spool file. HPrint is a handle to the print record for this printing operation. The name, volume reference number, and version number of the spool file will be taken from the job subrecord of this print record. After printing is successfully completed, the Printing Manager deletes the spool file from the disk.

PPrPort is a pointer to the storage to be used for the printing port for this operation. If this parameter is NIL, PrPicFile will allocate its own printing port. Similarly, pIOBuf points to an area of memory to be used as an input/output buffer for reading the spool file; if it's NIL, PrPicFile will use the volume buffer for the spool file's

volume. PDevBuf points to a similar buffer (the "band buffer") for holding the bit image to be printed; if NIL, PrPicFile will allocate its own buffer from the heap. As for PrOpenDoc, you'll normally want to set all of these storage parameters to NIL.

(note)
If you provide your own storage for pDevBuf, it has to be big enough to hold the number of bytes indicated by the iDevBytes field of the TPrXInfo subrecord of the print record.

(warning)
Be sure not to pass, in pPrPort, a pointer to the same printing port you received from PrOpenDoc, the one you originally used to spool the file. If that earlier port was allocated by PrOpenDoc itself (that is, if the pPrPort parameter to PrOpenDoc was NIL), then PrCloseDoc will have disposed of the port, making your pointer to it invalid. PrPicFile initializes a fresh printing port of its own; you just provide the storage (or let PrPicFile allocate it for itself). Of course, if you earlier provided your own storage to PrOpenDoc, there's no reason you can't use the same storage again for PrPicFile.

The prStatus parameter is a printer status record that PrPicFile will use to report on its progress. Your background procedure (if any) can use this record to monitor the state of the printing operation.

Handling Errors

FUNCTION PrError : INTEGER; [Pascal only]

PrError returns the result code returned by the last Printing Manager routine. The possible result codes are:

```
CONST noErr       = Ø;    {no error}
      iMemFullErr = -1Ø8; {not enough heap space}
```

and any Resource Manager result code. A result code of iMemFullErr means that the Memory Manager was unable to fulfill a memory allocation request by the Printing Manager.

PROCEDURE PrSetError (iErr: INTEGER); [Pascal only]

PrSetError stores the specified value into the global variable where the Printing Manager keeps its result code. The main *** (currently the only) *** use of this procedure is for canceling a printing operation in progress. To do this, write

```
PrSetError(iPrAbort)
```

where iPrAbort is the following predefined constant:

```
CONST iPrAbort = 128; {result code for halting printing}
```

Assembly-language note: You can achieve the same effect as PrSetError by storing directly into the location specified by printVars+iPrErr. *** Currently you shouldn't store into this location if it already contains an nonzero value. ***

Low-Level Driver Access

The routines in this section are used for communicating directly with the Printer Driver; the Printer Driver itself is described in the next section. You'll need to be familiar with the Device Manager to use the information given in this section.

```
PROCEDURE PrDrvrOpen;
```

PrDrvrOpen opens the Printer Driver.

```
PROCEDURE PrDrvrClose;
```

PrDrvrClose closes the Printer Driver.

```
PROCEDURE PrCtlCall (iWhichCtl: INTEGER; lParam1,lParam2,lParam3:
          LongInt);
```

PrCtlCall calls the Printer Driver's control routine. IWhichCtl designates the operation to be performed; the rest of the parameters depend on the operation.

```
FUNCTION PrDrvrDCE : Handle;
```

PrDrvrDCE returns a handle to the Printer Driver's device control entry.

```
FUNCTION PrDrvrVers : INTEGER;
```

PrDrvrVers returns the version number of the Printer Driver in the system resource file.

The version number of the Printing Manager is available as the predefined constant iPrRelease. You may want to compare the result of PrDrvrVers with iPrRelease to see if the Printer Driver in the resource file is the most recent version.

PROCEDURE PrNoPurge;

PrNoPurge prevents the Printer Driver from being purged from the heap.

PROCEDURE PrPurge;

PrPurge allows the Printer Driver to be purged from the heap.

THE PRINTER DRIVER

This section describes the Printer Driver, the device driver that communicates with a printer via the printer port or the modem port. Only programmers interested in low-level printing or writing their own device driver need read this. You'll need to be familiar with the Device Manager manual to use most of this information and the low-level routines described above.

The printer resource file for each variety of printer includes a device driver for that printer. When a particular printer is installed in the Printing Manager, the printer's device driver is copied from the printer resource file into the system resource file, making it the active Printer Driver.

The Printer Driver responds to the standard Device Manager calls OpenDriver, CloseDriver, Control, and Status. You can also communicate with it via the Printing Manager routines PrDrvrOpen, PrDrvrClose, and PrCtlCall. (The Status call is normally used only by the Font Manager.) Its driver name and driver reference number are available as the following predefined constants:

```
CONST sPrDrvr    = '.Print'; {Printer Driver resource name}
      iPrDrvrRef = -3;       {Printer Driver reference number}
```

To open the Printer Driver, call PrDrvrOpen; it'll remain open until you call PrDrvrClose. Calling PrNoPurge will prevent the driver from being purged from the heap until you call PrPurge.

You can call the PrDrvrVers function to determine whether the printing resources stored in the system resource file are compatible with the version of the Printing Manager you're using.

To get a handle to the driver's device control entry, call PrDrvrDCE. By calling the driver's control routine with PrCtlCall, you can perform a number of low-level printing operations such as bitmap printing, screen printing, and direct streaming of text to the printer (described

below). The first parameter to PrCtlCall, iWhichCtl, identifies the operation you want. The following values are predefined:

```
CONST iPrBitsCtl = 4; {bitMap printing}
      iPrIOCtl   = 5; {text streaming}
      iPrEvtCtl  = 6; {screen printing}
      iPrDevCtl  = 7; {device control}
      iFMgrCtl   = 8; {used by the Font Manager}
```

The remaining parameters of PrCtlCall--lParam1, lParam2, and lParam3--are three long integers whose meaning depends on the operation, as described below.

BitMap Printing

To send all or part of a bitMap directly to the printer, use PrCtlCall with iWhichCtl = iPrBitsCtl. Parameter lParam1 is a pointer to a QuickDraw bitMap; lParam2 is a pointer to the rectangle to be printed, in the coordinates of the printing port.

LParam3 is a printer-dependent parameter. On the Imagewriter it's used to control the printer's aspect ratio (the ratio of horizontal to vertical resolution). In low resolution, the Imagewriter normally prints 8Ø dots per inch horizontally by 72 vertically. This produces rectangular dots that are taller than they are wide. Since the Macintosh screen has square pixels (72 per inch both horizontally and vertically), images printed on the Imagewriter don't look exactly the same as they do on the screen.

To address this problem, the Imagewriter has a special square-dot mode that alters the speed of the print head to produce 72 dots per inch horizontally instead of 8Ø. Printing in this mode is slower than in the normal mode, but gives a more faithful reproduction of what the user sees on the screen. The user can choose which of the two modes to use by using the Printer program.

The value of the lParam3 parameter should be one of the following predefined constants:

```
CONST lScreenBits = Ø; {configurable}
      lPaintBits  = 1; {72 by 72 dots}
```

LScreenBits tells the Printer Driver to honor the user's selection between rectangular and square dots; lPaintBits overrides the user's choice and forces square dots.

Putting all this together, you can print the entire screen at the user's chosen aspect ratio with

```
PrCtlCall(iPrBitsCtl, ORD(@screenBits),
          ORD(@screenBits.bounds), lScreenBits)
```

To print the contents of a single window in square dots, use

```
PrCtlCall(iPrBitsCtl, ORD(@theWindow^.portBits),
          ORD(@theWindow^.portRect), lPaintBits)
```

Text Streaming

Text streaming is useful for fast printing of text when speed is more important than fancy formatting or visual fidelity. It gives you full access to the printer's native text facilities, such as control or escape sequences for boldface, italic, underlining, or condensed or expanded type, but makes no use of QuickDraw's elaborate formatting capabilities.

(warning)
Relying on specific printer capabilities and control sequences will make your application printer-dependent.

You can send a stream of text characters directly to the printer with iWhichCtl = iPrIOCtl. LParaml is a pointer to the beginning of the text; lParam2 is the number of bytes to transfer (a long integer); lParam3 is a pointer to an optional background procedure, or NIL for none.

IPrDevCtl is used for various printer control operations. When streaming text to the printer, you can use iPrDevCtl to perform these general operations in a printer-independent way, letting the Printer Driver take care of the details for a specific printer. The lParaml parameter specifies the operation you want:

```
CONST lPrReset    = $ØØØ1ØØØØ;  {reset printer}
      lPrPageEnd  = $ØØØ2ØØØØ;  {start new page}
      lPrLineFeed = $ØØØ3ØØØØ;  {start new line}
```

Before starting to print a document with text streaming, use

```
PrCtlCall(iPrDevCtl, lPrReset, Ø, Ø)
```

to reset the printer to its standard initial state. The parameters lParam2 and lParam3 are meaningless and should be set to Ø.

At the end of each printed line,

```
PrCtlCall(iPrDevCtl, lPrLineFeed, Ø, Ø)
```

advances the paper one line and returns to the left margin. This achieves the effect of the standard "CRLF" (carriage-return-line-feed) sequence in a printer-independent way. It's strongly recommended that you use this method instead of sending carriage returns and line feeds directly to the printer. The lParam2 parameter tells how far to advance the paper; lParam3 is meaningless and should be set to Ø.
*** The exact use of lParam2 in this call hasn't yet been determined.

A value of Ø will probably denote the printer's standard line height, which is usually what you'll want. ***

At the end of each page,

```
PrCtlCall(iPrDevCtl, lPrPageEnd, Ø, Ø)
```

does whatever is appropriate for the given printer, such as sending a form feed character and advancing past the paper fold. It's recommended that you use this call instead of just sending a form feed yourself. LParam2 and lParam3 are meaningless and should be set to Ø.

Screen Printing

IPrEvtCtl does an immediate dump of all or part of the screen directly to the printer. LParaml is one of the following codes:

```
CONST iPrEvtAll = $ØØØ2FFFD;    {print whole screen}
      iPrEvtTop = $ØØØ1FFFD;    {print top (frontmost) window}
```

The other two parameters are meaningless and should be set to Ø. So, for example,

```
PrCtlCall(iPrEvtCtl, iPrEvtAll, Ø, Ø)
```

prints the entire screen at the user's chosen aspect ratio, and

```
PrCtlCall(iPrEvtCtl, iPrEvtTop, Ø, Ø)
```

prints just the frontmost window.

The Operating System Event Manager uses this call to do immediate screen printing when the user types a special key combination (Command-$ for the frontmost window, the same with Caps Lock for the full screen).

Font Manager Support

The Printer Driver provides one Status and one Control call for use by the Font Manager in selecting fonts for a given printer. Both are identified by the following csCode value

```
CONST iFMgrCtl = 8;
```

With the Status call, the Font Manager asks for the printer's font characterization table. After using the information in this table to select a font, it issues the Control call to give the Printer Driver a chance to modify the choice. This process is described further in the Font Manager manual.

PRINTING RESOURCES

For programmers who want to write their own device drivers for different printers or modify existing drivers, this section describes the two files that contain the resources needed to run the Printing Manager: the system resource file and the printer resource file (see Figure 7). Most of the data described in this section is accessible only to assembly-language programmers.

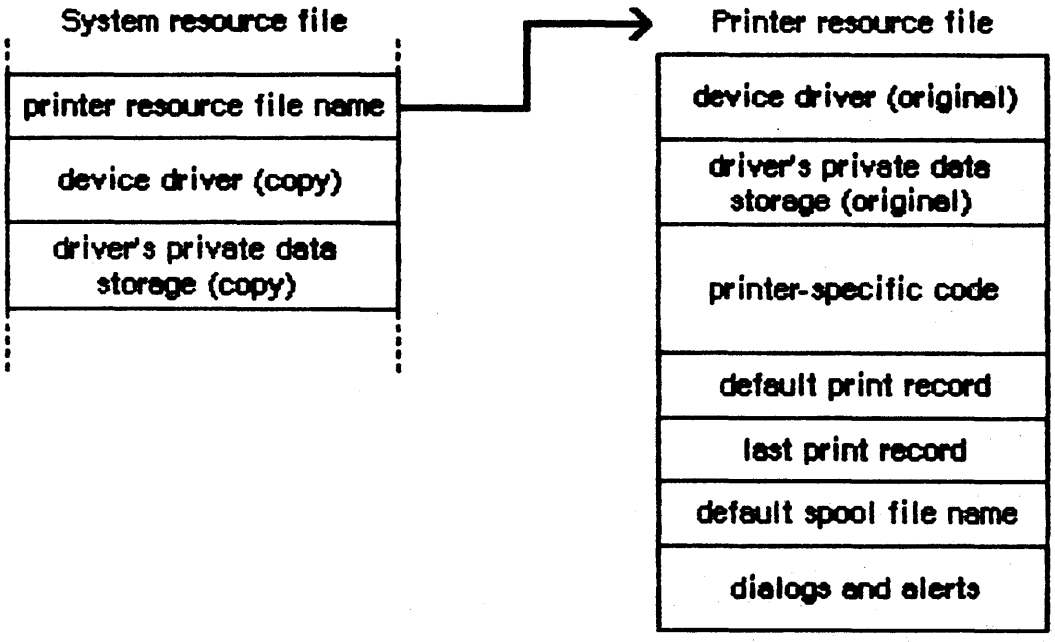


Figure 7. Printing Resources

The system resource file contains:

Resource	Resource type	Resource ID
Name of the current printer resource file	'STR '	$E000
A copy of the device driver for the currently installed printer	'DRVR'	2
A copy of the driver's private data storage	'PREC'	2

The printer resource file contains the following information:

Resource	Resource type	Resource ID
The device driver for this printer	'DRVR'	$EØØØ
The driver's private storage	'PREC'	$EØØØ
Printer-specific code used to implement Printing Manager routines	'PDEF'	Ø through 6 (see below)
Default print record for use with this printer	'PREC'	Ø
Print record from the previous printing operation	'PREC'	1
Default spool file name	'STR '	$EØØ1
Style dialog	'DLOG'	$EØØØ
Job dialog	'DLOG'	$EØØ1
Installation dialog	'DLOG'	$EØØ2
Alerts	'ALRT'	(private)
Dialog and alert item lists	'DITL'	(private)

Notice that the Printer Driver and its private storage are kept in both the system and printer resource files. The copies in the system resource file are the ones actually used; those in the printer resource file are there just to be copied into the system resource file when a new printer is installed. Installing a new printer is done by copying the driver and its private storage from the printer resource file to the system resource file and placing the name of the printer resource file in the system resource file. (You can use this method to install a printer yourself, but normally it's done by the Printer program at the user's request.)

You can use the following predefined constants to identify the various resource types and IDs in the printer resource file (they'll be different in the system resource file):

```
CONST lPrintType = $0524543; {type ('PREC') for print records and }
                             { private storage}
      iPrintDef  = 0;        {ID for default print record}
      iPrintLst  = 1;        {ID for previous print record}
      iPrintDrvr = 2;        {ID for Printer Driver and its private }
                             { storage in system resource }
                             { file}
      iMyPrDrvr = $E000;     {ID for Printer Driver and its private }
                             { storage }

      iPStrRFil  = $E000;    {ID for printer resource file name}
      iPStrPFil  = $E001;    {ID for default spool file name}

      iPrStlDlg  = $E000;    {ID for style dialog}
      iPrJobDlg  = $E001;    {ID for job dialog}
```

The most important items in a printer resource file are the Printer Driver and the printer-specific code. The driver has the standard structure for device drivers, as described in the Device Manager manual, and implements the Control and Status calls as discussed above under "The Printer Driver".

The printer-specific code is kept in a series of separate overlays. They are all of resource type 'PDEF', and their resource IDs are available to assembly-language programmers as the following predefined constants:

```
iPrDraftID    .EQU   0 ;draft printing
iPrSpoolID    .EQU   1 ;spooling
iPrUser1ID    .EQU   2 ;printer-specific printing, method 1
iPrUser2ID    .EQU   3 ;printer-specific printing, method 2
iPrDlgsID     .EQU   4 ;print records and dialogs
iPrPicID      .EQU   5 ;spool printing
```

Overlays 0 and 1 do draft printing and spooling, respectively; overlays 2 and 3, if present, provide additional printing methods for a particular printer. All four overlays include the same routines, but implement them in different ways for the different printing methods. When one of the routines is called, the Printing Manager uses the bJDocLoop field in the job subrecord to decide which overlay to use. Each overlay begins with a list of offsets to the locations of the routines within that overlay.

```
lOpenDoc      .EQU       $000C0000     ;PrOpenDoc
lCloseDoc     .EQU       $00048004     ;PrCloseDoc
lOpenPage     .EQU       $00080008     ;PrOpenPage
lClosePage    .EQU       $0004000C     ;PrClosePage
```

This list is followed by the code of the routines themselves.

Overlay 4 contains the Printing Manager's routines for manipulating print records and dialogs:

```
lDefault    .EQU        $00048000     ;PrintDefault
lStlDialog  .EQU        $00048004     ;PrStlDialog
lJobDialog  .EQU        $00048008     ;PrJobDialog
lStlInit    .EQU        $0004000C     ;PrStlInit
lJobInit    .EQU        $00040010     ;PrJobInit
lDlgMain    .EQU        $00048014     ;PrDlgMain
lValidate   .EQU        $00048018     ;PrValidate
lJobMerge   .EQU        $0008801C     ;PrJobMerge
```

*** PrStlInit, PrJobInit, and PrDlgMain are used in customizing the dialogs, and will be covered in a later draft of this manual. ***

Overlays 5 contains just the spool-printing routine PrPicFile (it's still preceded by an offset, however):

```
lPrPicFile  .EQU        $00148000     ;PrPicFile
```

SUMMARY OF THE PRINTING MANAGER

Constants

```
CONST { Result codes }

      iMemFullErr = -1Ø8; {not enough heap space}
      noErr       =    Ø; {no error}

      { Printing methods }

      bDraftLoop = Ø;   {draft printing}
      bSpoolLoop = 1;   {spooling}
      bUser1Loop = 2;   {printer-specific, method 1}
      bUser2Loop = 3;   {printer-specific, method 2}

      { Printer Driver Control call parameters }

      iPrBitsCtl  = 4;          {bitMap printing}
      lScreenBits = Ø;            {configurable}
      lPaintBits  = 1;            {72 by 72 dots}
      iPrIOCtl    = 5;          {text streaming}
      iPrEvtCtl   = 6;          {screen printing}
      iPrEvtAll   = $ØØØ2FFFD;    {print whole screen}
      iPrEvtTop   = $ØØØ1FFFD;    {print top (frontmost) window}
      iPrDevCtl   = 7;          {device control}
      lPrReset    = $ØØØ1ØØØØ;    {reset printer}
      lPrPageEnd  = $ØØØ2ØØØØ;    {start new page}
      lPrLineFeed = $ØØØ3ØØØØ;    {start new line}
      iFMgrCtl    = 8;          {used by the Font Manager}

      { Miscellaneous }

      iPFMaxPgs  = 128;       {maximum number of pages in a spool file}
      iPrPgFract = 12Ø;       {units per inch of paper dimension}
      iPrAbort   = 128;       {result code for halting printing}
      iPrRelease = 2;         {current version number of Printing }
                              { Manager}
      lPfType    = $5Ø46484C; {spool file type 'PFIL'}
      lPfSig     = $5Ø535953; {spool file creator 'PSYS'}

      { Printing resources }

      sPrDrvr    = '.Print';  {Printer Driver resource name}
      iPrDrvrRef = -3;        {Printer Driver reference number}
      lPrintType = $5Ø524543; {type ('PREC') for print records }
                              { and private storage}
      iPrintDef  = Ø;         {ID for default print record}
      iPrintLst  = 1;         {ID for previous print record}
      iPrintDrvr = 2;         {ID for Printer Driver and its }
                              { private storage in system }
```

```
                                { resource file}
    iMyPrDrvr = $E000;          {ID for Printer Driver and its }
                                { private storage in printer }
                                { resource file}
    iPStrRFil  = $E000;         {ID for printer resource file name}
    iPStrPFil  = $E001;         {ID for default spool file name}
    iPrStlDlg  = $E000;         {ID for style dialog}
    iPrJobDlg  = $E001;         {ID for job dialog}
```

Data Types

```
TYPE TPStr80 = ^TStr80;
     TStr80  = STRING[80];

     TPRect  = ^Rect;

     TPPrPort = ^TPrPort;
     TPrPort  = RECORD
                  gPort:  GrafPort; {grafPort to draw in}
                  gProcs: QDProcs;  {pointers to drawing routines}
                  {more fields for internal use only}
                END;

     TPPort = PACKED RECORD
                CASE INTEGER OF
                  0: (pGPort:  GrafPtr);
                  1: (pPrPort: TPPrPort)
              END;

     THPrint = ^TPPrint;
     TPPrint = ^TPrint;
     TPrint  = RECORD
                 iPrVersion: INTEGER;  {Printing Manager version}
                 prInfo:     TPrInfo;  {printer information}
                 rPaper:     Rect;     {paper rectangle}
                 prStl:      TPrStl;   {style information}
                 prInfoPT:   TPrInfo;  {copy of PrInfo}
                 prXInfo:    TPrXInfo; {band information}
                 prJob:      TPrJob;   {job information}
                 printX:     ARRAY [1..19] OF INTEGER
                                       {used internally}
               END;

     TPrInfo = RECORD
                 iDev:  INTEGER; {driver information}
                 iVRes: INTEGER; {printer vertical resolution}
                 iHRes: INTEGER; {printer horizontal resolution}
                 rPage: Rect     {page rectangle}
               END;
```

```
TPrStl = RECORD
           wDev:   TWord;       {used internally}
           iPageV: INTEGER;     {paper height}
           iPageH: INTEGER;     {paper width}
           bPort:  SignedByte; {printer or modem port}
           feed:   TFeed        {paper type}
         END;

TFeed = (feedCut,      {hand-fed, individually cut sheets}
         feedFanfold, {continuous-feed fanfold paper}
         feedMechCut, {mechanically fed cut sheets}
         feedOther);  {other types of paper}

TPrJob  = RECORD
            iFstPage:  INTEGER;     {first page to print}
            iLstPage:  INTEGER;     {last page to print}
            iCopies:   INTEGER;     {number of copies}
            bJDocLoop: SignedByte; {printing method}
            fFromUsr:  BOOLEAN;  {TRUE if called from application}
            pIdleProc: ProcPtr;     {background procedure}
            pFileName: TPStr8Ø;     {spool file name}
            iFileVol:  INTEGER;     {volume reference number}
            bFileVers: SignedByte; {version number of spool file}
            bJobX:     SignedByte  {not used}
          END;

TPrXInfo = RECORD
             iRowBytes: INTEGER;     {bytes per row}
             iBandV:    INTEGER;     {vertical dots}
             iBandH:    INTEGER;     {horizontal dots}
             iDevBytes: INTEGER;     {size of bit image}
             iBands:    INTEGER;     {bands per page}
             bPatScale: SignedByte; {used by QuickDraw}
             bUlThick:  SignedByte; {underline thickness}
             bUlOffset: SignedByte; {underline offset}
             bUlShadow: SignedByte; {underline descender}
             scan:      TScan;       {scan direction}
             bXInfoX:   SignedByte  {not used}
           END;

TScan = (scanTB,  {scan top to bottom}
         scanBT,  {scan bottom to top}
         scanLR,  {scan bottom to top}
         scanRL); {scan right to left}
```

```
TPrStatus = RECORD
              iTotPages:  INTEGER;  {total number of pages}
              iCurPage:   INTEGER;  {page being printed}
              iTotCopies: INTEGER;  {number of copies}
              iCurCopy:   INTEGER;  {copy being printed}
              iTotBands:  INTEGER;  {bands per page}
              iCurBand:   INTEGER;  {band being printed}
              fPgDirty:   BOOLEAN;  {TRUE if started printing page}
              fImaging:   BOOLEAN;  {TRUE if imaging}
              hPrint:     THPrint;  {print record}
              pPrPort:    TPPrPort; {printing port}
              hPic:       PicHandle {used internally}
            END;
```

Routines

Initialization and Termination

```
PROCEDURE PrOpen;
PROCEDURE PrClose;
```

Print Records and Dialogs

```
PROCEDURE PrintDefault (hPrint: THPrint);
FUNCTION  PrValidate   (hPrint: THPrint) : BOOLEAN;
FUNCTION  PrStlDialog  (hPrint: THPrint) : BOOLEAN;
FUNCTION  PrJobDialog  (hPrint: THPrint) : BOOLEAN;
PROCEDURE PrJobMerge   (hPrintSrc,hPrintDst: THPrint);
```

Document Printing

```
FUNCTION  PrOpenDoc   (hPrint: THPrint; pPrPort: TPPrPort; pIOBuf: Ptr) :
                       TPPrPort;
PROCEDURE PrCloseDoc  (pPrPort: TPPrPort);
PROCEDURE PrOpenPage  (pPrPort: TPPrPort; pPageFrame: TPRect);
PROCEDURE PrClosePage (pPrPort: TPPrPort);
```

Spool Printing

```
PROCEDURE PrPicFile (hPrint: THPrint; pPrPort: TPPrPort; pIOBuf: Ptr;
                     pDevBuf: Ptr; VAR prStatus: TPrStatus);
```

Handling Errors [Pascal only]

```
FUNCTION  PrError :   INTEGER;
PROCEDURE PrSetError (iErr: INTEGER);
```

Low-Level Driver Access

```
PROCEDURE PrDrvrOpen;
PROCEDURE PrDrvrClose;
PROCEDURE PrCtlCall   (iWhichCtl: INTEGER; lParam1,lParam2,lParam3:
                       LongInt);
FUNCTION  PrDrvrDCE :  Handle;
FUNCTION  PrDrvrVers : INTEGER;
PROCEDURE PrNoPurge;
PROCEDURE PrPurge;
```

Resource File Contents

System Resource File

Resource	Resource type	Resource ID
Name of the current printer resource file	'STR '	-8192
A copy of the device driver for the currently installed printer	'DRVR'	2
A copy of the driver's private data storage	'PREC'	2

Printer Resource File

Resource	Resource type	Resource ID
Original copy of the device driver for this printer	'DRVR'	-8192
Original copy of the driver's private storage	'PREC'	-8192
Printer-specific code used to implement Printing Manager routines	'PDEF'	Ø through 6
Default print record for use with this printer	'PREC'	Ø
Print record from the previous printing operation	'PREC'	1
Default spool file name	'STR '	-8191
Style dialog	'DLOG'	-8192
Job dialog	'DLOG'	-8191
Installation dialog	'DLOG'	-819Ø
Alert definitions	'ALRT'	(private)
Dialog and alert item lists	'DITL'	(private)

Assembly-Language Information

Constants

```
; Result codes

iMemFullErr    .EQU     -1Ø8    ;not enough heap space
noErr          .EQU        Ø    ;no error

; Printing methods

bDraftLoop     .EQU     Ø    ;draft printing
bSpoolLoop     .EQU     1    ;spooling
bUser1Loop     .EQU     2    ;printer-specific, method 1
bUser2Loop     .EQU     3    ;printer-specific, method 2

; Printer Driver Control call parameters

iPrBitsCtl     .EQU     4            ;bitMap printing
```

```
lScreenBits     .EQU    Ø               ;  configurable
lPaintBits      .EQU    1               ;  72 by 72 dots
iPrIOCtl        .EQU    5               ;text streaming
iPrEvtCtl       .EQU    6               ;screen printing
iPrEvtAll       .EQU    $ØØFFFFFD       ;  print whole screen
iPrEvtTop       .EQU    $ØØFEFFFD       ;  print top (frontmost) window
iPrDevCtl       .EQU    7               ;device control
lPrReset        .EQU    1               ;  reset printer
lPrPageEnd      .EQU    2               ;  start new page
lPrLineFeed     .EQU    3               ;  start new line
iFMgrCtl        .EQU    8               ;used by the Font Manager

; Miscellaneous

iPrintSize      .EQU    12Ø         ;length of print record
iPrPortSize     .EQU    178         ;length of printing port
iPrStatSize     .EQU    26          ;length of printer status record
iPrAbort        .EQU    128         ;result code for halting printing
iPrRelease      .EQU    2           ;current version number of Printing
                                    ; Manager
lPfType         .EQU    $5Ø46484C   ;file type ('PFIL') for spool files
lPfSig          .EQU    $5Ø535953   ;signature ('PSYS') of Printer program

; Printing resources

iPrDrvrRef      .EQU    -3          ;Printer Driver reference number
lPrintType      .EQU    $5Ø524543   ;type ('PREC') for print records
                                    ; and private storage
iPrintDef       .EQU    Ø           ;ID for default print record
iPrintLst       .EQU    1           ;ID for previous print record
iPrDrvrID       .EQU    2           ;ID for Printer Driver and its
                                    ; private storage in system
                                    ; resource file
lPStrType       .EQU    $53545220   ;type 'STR ' for file name
                                    ; resources
iPStrRFil       .EQU    $EØØØ       ;ID for printer resource file
                                    ; name
iPStrPFil       .EQU    $EØØ1       ;ID for default spool file name
iPrStlDlg       .EQU    $EØØØ       ;ID for style dialog
iPrJobDlg       .EQU    $EØØ1       ;ID for job dialog

; Resource IDs for code overlays

iPrDraftID      .EQU        Ø       ;draft printing
iPrSpoolID      .EQU        1       ;spooling
iPrUser1ID      .EQU        2       ;printer-specific printing, method 1
iPrUser2ID      .EQU        3       ;printer-specific printing, method 2
iPrDlgsID       .EQU        4       ;print records and dialogs
iPrPicID        .EQU        5       ;spool printing
```

```
; Offsets to document printing code overlays

lOpenDoc      .EQU        $000C0000     ;PrOpenDoc
lCloseDoc     .EQU        $00048004     ;PrCloseDoc
lOpenPage     .EQU        $00080008     ;PrOpenPage
lClosePage    .EQU        $0004000C     ;PrClosePage

; Offsets to print record and dialog code overlays

lDefault      .EQU        $00048000     ;PrintDefault
lStlDialog    .EQU        $00048004     ;PrStlDialog
lJobDialog    .EQU        $00048008     ;PrJobDialog
lStlInit      .EQU        $0004000C     ;PrStlInit
lJobInit      .EQU        $00040010     ;PrJobInit
lDlgMain      .EQU        $00048014     ;PrDlgMain
lValidate     .EQU        $00048018     ;PrValidate
lJobMerge     .EQU        $0008801C     ;PrJobMerge

; Offset to spool printing code overlay

lPrPicFile    .EQU        $00148000     ;PrPicFile
```

Printing Port

```
gPort         GrafPort to draw in
gProcs        Pointers to drawing routines
```

Print Record

```
iPrVersion    Printing Manager version
prInfo        Printer information
rPaper        Paper rectangle
prStl         Style information
prJob         Job information
```

Printer Information Subrecord

```
iDev          Driver information
iVRes         Printer vertical resolution
iHRes         Printer horizontal resolution
rPage         Page rectangle
```

Style Subrecord

```
iPageV        Paper height
iPageH        Paper width
bPort         Printer or modem port
feed          Paper type
```

Job Subrecord

iFstPage	First page to print
iLstPage	Last page to print
iCopies	Number of copies
bJDocLoop	Printing method
fFromApp	Nonzero if called from application
pIdleProc	Pointer to background procedure
pFileName	Spool file name
iFileVol	Volume reference number
bFileVers	Version number spool file

Band Information Subrecord

iRowBytes	Bytes per row
iBandV	Vertical dots
iBandH	Horizontal dots
iDevBytes	Size of bit image
iBands	Bands per page
bPatScale	Used by QuickDraw
bUlThick	Underline thickness
bUlOffset	Underline offset
bUlShadow	Underline descender
scan	Scan direction

Printer Status Record

iTotPages	Total number of pages
iCurPage	Page being printed
iTotCopies	Number of copies
iCurCopy	Copy being printed
iTotBands	Bands per page
iCurBand	Band being printed
fPgDirty	Nonzero if started printing page
fImaging	Nonzero if imaging
hPrint	Print record
pPrPort	Printing port

Variables

Name	Size	Contents
printVars+iPrErr	2 bytes	Current result code

GLOSSARY

background procedure: A procedure passed to the Printing Manager to be run during idle times in the printing process.

band: One of the sections into which a page is divided for imaging and printing.

draft printing: Printing a document by using QuickDraw calls to drive the printer's character generator directly.

imaging: The process of converting an application's description of an image (such as a QuickDraw picture) into an actual array of bits to be displayed or printed.

job dialog: A dialog pertaining to one particular printing job; conventionally associated with the application's Print command.

landscape orientation: The positioning of a document in a printer with the long dimension of the paper running horizontally.

page rectangle: The rectangle marking the boundaries of a printed page image.

paper rectangle: The rectangle marking the boundaries of the physical sheet of paper on which a page is printed.

portrait orientation: The positioning of a document in a printer with the long dimension of the paper running vertically.

Printer: A special application program for printing spool files from a disk and configuring different printers.

Printer Driver: The device driver for the currently installed printer.

printer resource file: A file containing all the resources needed to run the Printing Manager with a particular printer.

printer status record: A record used by the Printing Manager to report on the progress of printing operations.

printing port: A special grafPort customized for printing instead of drawing on the screen.

print record: A record containing all the information needed by the Printing Manager to perform a particular printing job.

spool file: A disk file created as the result of spooling.

spooling: Writing a representation of a document's printed image to a disk file, rather than directly to the printer.

spool printing: Printing the image contained in a spool file.

style dialog: A dialog pertaining to the use of the printer for a particular document; conventionally associated with the application's Page Setup command.

MACINTOSH USER EDUCATION

The Memory Manager: A Programmer's Guide /MEM.MGR/MEMORY

See Also: The Resource Manager: A Programmer's Guide

Modification History: First Draft (ROM 7) S. Chernicoff 1Ø/1Ø/83

ABSTRACT

This manual describes the Memory Manager, the part of the Macintosh Operating System that controls the dynamic allocation of memory space on the heap.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Memory Manager, the part of the Macintosh Operating System that controls the dynamic allocation of memory space on the heap. *** Eventually it will become part of a larger manual describing the entire Operating System. ***

(eye)

> This manual describes version 7, the final, "frozen" version of the Macintosh ROM. Earlier versions may not work exactly as described here. *** There may someday be one or more special, RAM-based versions of the Memory Manager for software development purposes, doing more extensive error checking or gathering statistics on a program's memory usage. This manual describes the ROM-based version only. ***

Like all Operating System documentation, this manual is intended for both Pascal and assembly-language programmers. All readers are assumed to be familiar with Lisa Pascal; information of interest only to assembly-language programmers is isolated and labeled so that Pascal programmers can conveniently skip it. Whichever is your preferred language, please bear with occasional remarks addressed solely to the other group.

The manual begins with an introduction to the Memory Manager and what it's used for. It then discusses some basic concepts behind the Memory Manager's operation: how blocks of memory are allocated within the heap and how the allocated blocks are referred to by programs that use them. Following this is a discussion of the internal data structures that the Memory Manager uses to find its way around in the heap.

A section on using the Memory Manager introduces its routines and tells how they fit into the flow of your application program. This is followed by detailed descriptions of all Memory Manager procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that will not be of interest to all readers. Special information is given on unusual techniques that you may find useful in working with the Memory Manager and on how to use it from assembly-language programs.

Finally, there is a quick-reference summary of the Memory Manager's data structures and routines, along with a glossary of terms used in this manual.

ABOUT THE MEMORY MANAGER

Using the Memory Manager, your program can maintain one or more independent areas of heap memory (called heap zones) and use them to allocate blocks of memory of any desired size. Unlike stack space, which is always allocated and released in strict LIFO (last-in-first-out) order, blocks on the heap can be allocated and released in any order, according to your program's needs. So instead of growing and shrinking in an orderly way like the stack, the heap tends to become fragmented into a patchwork of allocated and free blocks, as shown in Figure 1. The Memory Manager does all the necessary "housekeeping" to keep track of the blocks as it allocates and releases them.

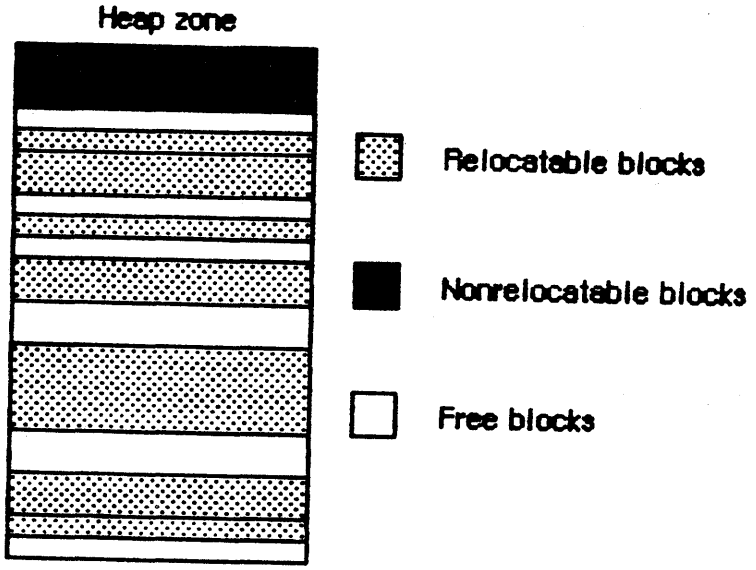


Figure 1. A Fragmented Heap

All memory allocation is performed within a particular heap zone. The Memory Manager always maintains at least two heap zones: a system heap zone, reserved for the system's own use, and an application heap zone for use by your program. The system heap zone is initialized to 16K bytes when the system is started up. Objects in this zone remain allocated even when one application terminates and another is launched. The application heap zone is automatically reinitialized at the start of each new application program, and the contents of any previous application zone are lost. The initial size of the application zone is 6K bytes, but it can grow as needed to create more heap space while the program is running. Your program can create additional heap zones if it chooses, either by subdividing this original application zone or by allocating space on the stack for more heap zones.

(hand)

In this manual, unless otherwise stated, the term "application heap zone" (or just "application zone")

always refers to the original application heap zone provided by the system, before any subdivision.

Various parts of the Macintosh Operating System and Toolbox also use space in the application heap zone. For instance, the actual machine-language code of your program resides in the application zone, in space reserved for it at the request of the Segment Loader. Similarly, the Resource Manager requests space in the application zone to hold resources it has read into memory from a resource file. Toolbox routines that create new entities of various kinds, such as NewWindow, NewControl, and NewMenu, implicitly call the Memory Manager to allocate the space they need.

At any given time, there is exactly one current heap zone, to which most Memory Manager operations implicitly apply. You can control which heap zone is current by calling a Memory Manager procedure. Whenever the system needs to access its own (system) heap zone, it saves the setting of the current heap zone and restores it later, so that the operation is transparent to your program.

Space within a heap zone is divided up into contiguous pieces called blocks. The blocks in a zone fill it completely: every byte in the zone is part of exactly one block, which may be either allocated (reserved for use by your program or by the system) or free (available for allocation). Each block has a block header containing information for the Memory Manager's own use, followed by the block's contents, the area available for use (see Figure 2). There may also be some unused bytes at the end of the block, beyond the end of the contents.

Assembly-language note: Blocks are always aligned on even word boundaries, so you can access them with word (.W) and long-word (.L) instructions.

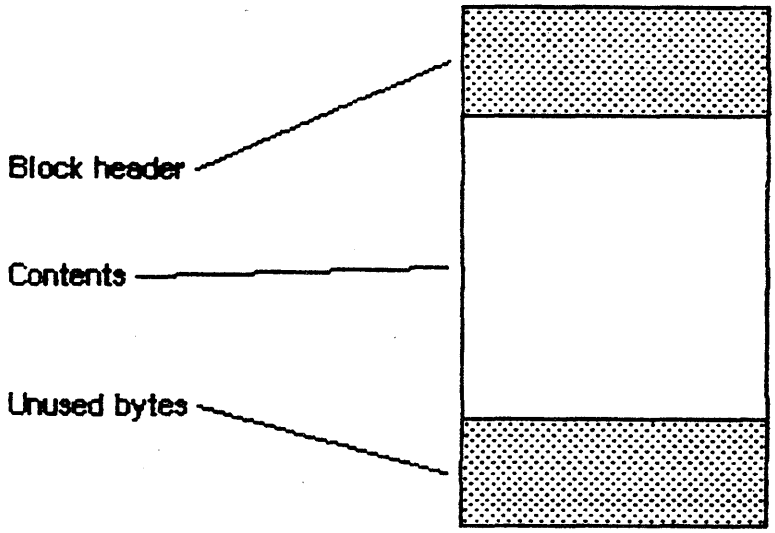


Figure 2. A Block

A block can be of any size, limited only by the size of the heap zone itself. What's inside the block is of no concern to the Memory Manager: it may contain data being used by your program, executable code forming part of the program itself, resource information read from a resource file, or anything else that may be appropriate. To the Memory Manager, it's just a block of a certain size.

(hand)

Don't confuse the blocks manipulated by the Memory Manager with disk blocks, which are always 512 bytes long.

An allocated block may be relocatable or nonrelocatable; if relocatable, it may be locked or unlocked; if unlocked, it may be purgeable or unpurgeable. Relocatable blocks can be moved around within the heap zone to create space for other blocks; nonrelocatable blocks can never be moved. These are permanent properties of a block that can never be changed once the block is allocated. The remaining attributes (locked and unlocked, purgeable and unpurgeable) can be set and changed as necessary. Locking a relocatable block prevents it from being moved, but only temporarily: you can unlock the block at any time, again allowing the Memory Manager to move it. Making a block purgeable allows the Memory Manager to remove it from the heap zone, if necessary, to make room for another block. (Purging of blocks is discussed further below under "How Heap Space Is Allocated".) A newly allocated block is initially unlocked and unpurgeable.

POINTERS AND HANDLES

Relocatable and nonrelocatable blocks are referred to in different ways: nonrelocatable blocks by pointers, relocatable blocks by handles (discussed below). When the Memory Manager allocates a new nonrelocatable block, it returns a pointer to the block. Thereafter, whenever you need to refer to the block, you use this pointer. Like any other pointer, it's simply a memory address: that of the first byte in the block's contents (see Figure 3). You can make as many copies of this pointer as you like. Since the block they point to can never be moved within its heap zone, you can rely on all copies of the pointer to remain correct. They will continue to point to the block for as long as the block remains allocated.

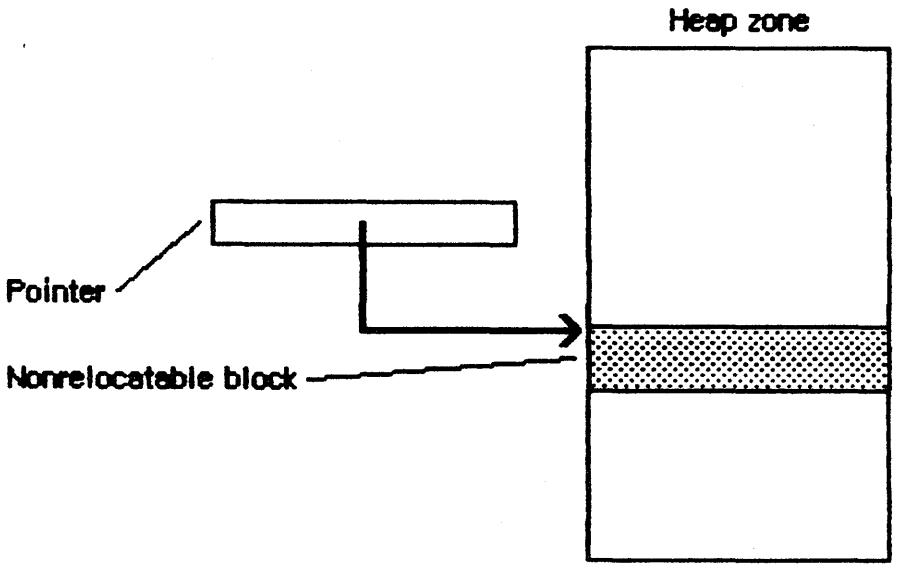


Figure 3. A Pointer to a Nonrelocatable Block

Relocatable blocks don't share this property, however. If necessary to make room for some other block, the Memory Manager can move a relocatable block at any time to a new location in its heap zone. This would leave any pointers you might have to the block pointing to the wrong place in memory, or "dangling". Dangling pointers can be very difficult to diagnose and correct, since their effects typically aren't discovered until long after the pointer is left dangling.

To help avoid dangling pointers, the Memory Manager maintains a single master pointer to each relocatable block, allocated from within the same heap zone as the block itself. The master pointer is created at the same time as the block and set to point to it. What you get back from the Memory Manager when you allocate a relocatable block is a pointer to the master pointer, called a handle to the block (see Figure 4). From then on, you always use this handle to refer to the block. If the Memory Manager later has to move the block, it has only to

update the master pointer to point to the block's new location; the master pointer itself is never moved. Since all copies of the handle point to the block by double indirection through this same master pointer, they can be relied on not to dangle, even after the block has been moved.

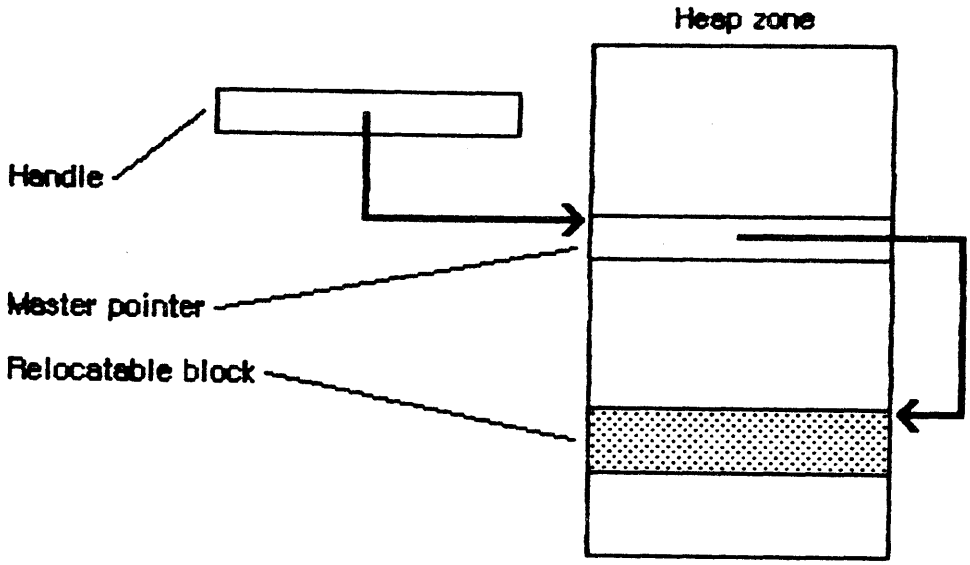


Figure 4. A Handle to a Relocatable Block

(eye)

To maintain the integrity of the memory allocation system, always use the Memory Manager routines provided (or other Operating System or Toolbox routines that call them) to allocate and release space on the heap. Don't use the Pascal standard procedures NEW and DISPOSE. *** Eventually the versions of these routines in the Pascal Library will be changed to work through the Memory Manager. ***

HOW HEAP SPACE IS ALLOCATED

The Memory Manager allocates space in a heap zone according to a "first fit" strategy. When you ask to allocate a block of a certain size, the Memory Manager scans the current heap zone looking for a place to put the new block. For relocatable blocks, it looks for a free block of at least the requested size, scanning forward from the end of the last block allocated and "wrapping around" if necessary from the end of the zone to the beginning. (Nonrelocatable blocks are handled a bit differently, as described below.) As soon as it finds a free block big enough, it allocates the requested number of bytes from that block. That is, it uses the first free block it finds that's big enough to satisfy the request, instead of continuing to search for a better fit.

If a single free block can't be found that's big enough, the Memory Manager tries to create one by compacting the heap zone: moving allocated blocks together in order to collect the free space into a single larger free block (see Figure 5). Only relocatable, unlocked blocks can be moved. The compaction continues until either a free block of at least the requested size has been created or the entire heap zone has been compacted.

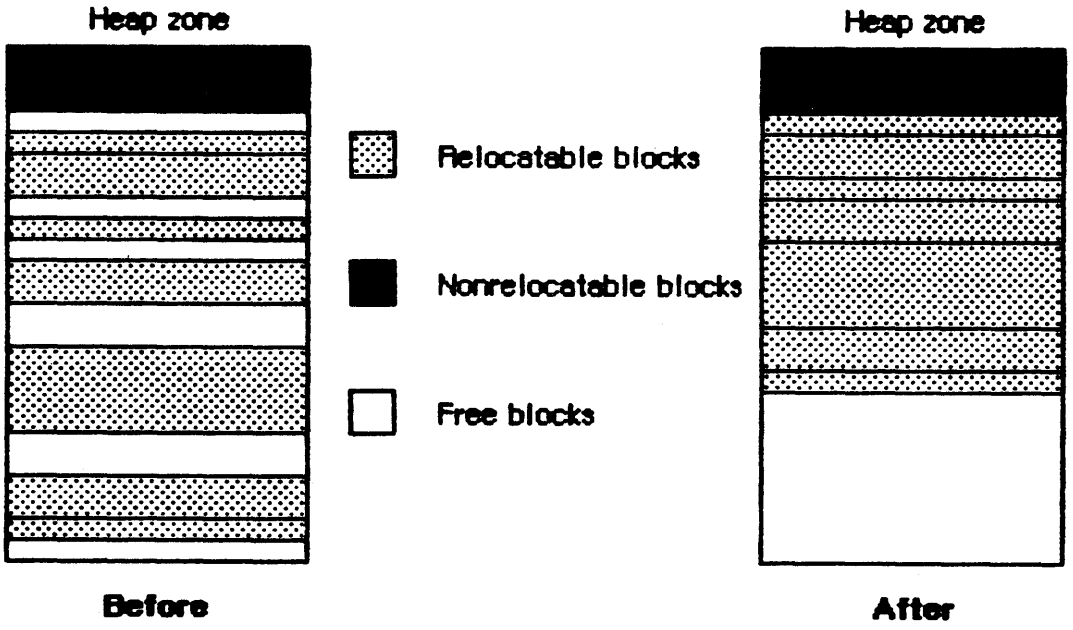


Figure 5. Heap Compaction

Notice that nonrelocatable blocks (and relocatable ones that are temporarily locked) tend to interfere with the compaction process by forming immovable "islands" in the heap. This can prevent free blocks from being collected together and lead to fragmentation of the available free space, as shown in Figure 6. To minimize this problem, the Memory Manager tries to keep all the nonrelocatable blocks together at the beginning of the heap zone. When you allocate a nonrelocatable block, the Memory Manager will do everything in its power to make room for the new block at the lowest available position in the zone, including moving other blocks upward, expanding the zone, or purging blocks from it (see below).

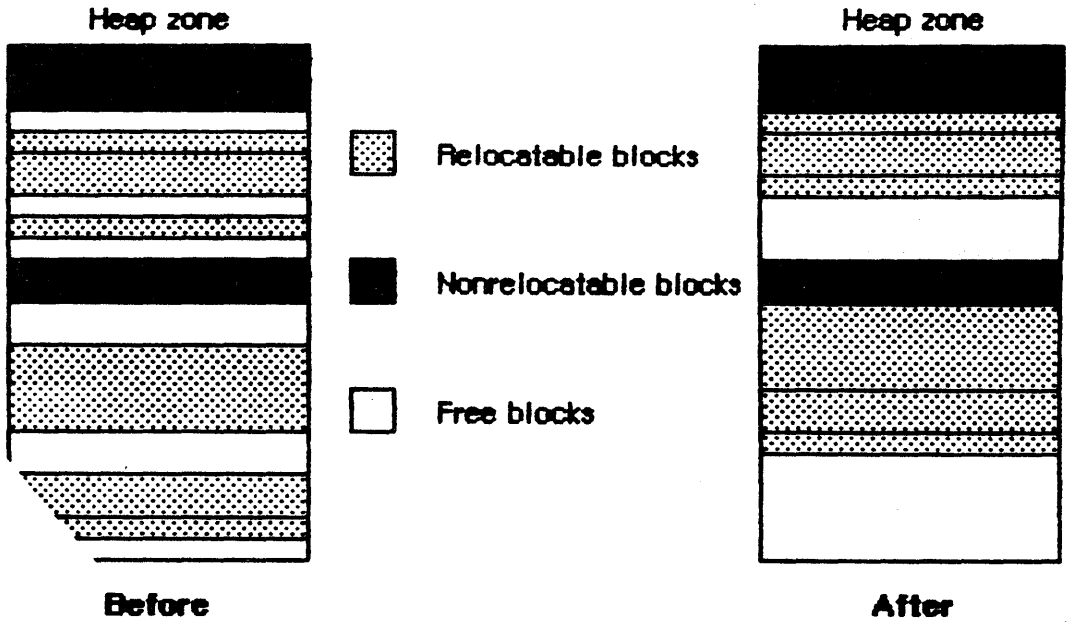


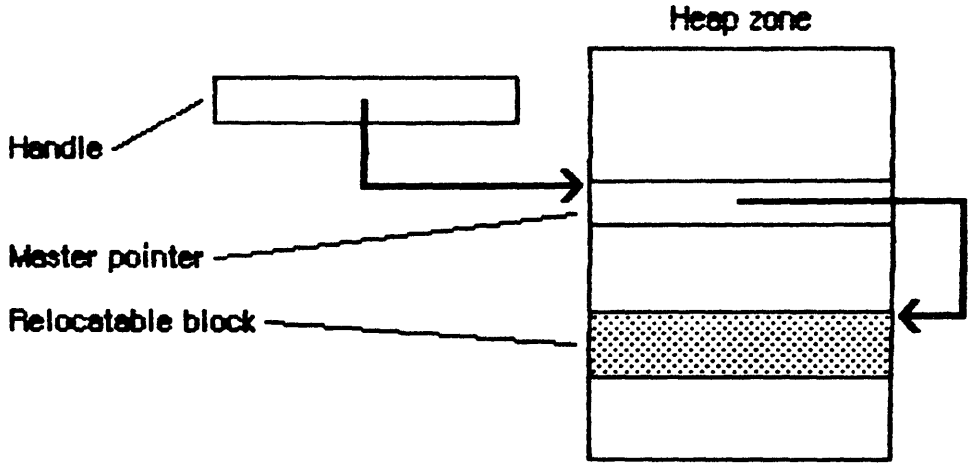
Figure 6. Fragmentation of Free Space

If the Memory Manager still can't satisfy the allocation request after compacting the entire heap zone, it next tries expanding the zone by the requested number of bytes, rounded upward to the nearest 1K. Only the original application zone can be expanded, and only up to a certain limit (discussed more fully under "The Stack and the Heap", below). If any other zone is current, or if the application zone has already reached or exceeded its limit, this step is skipped.

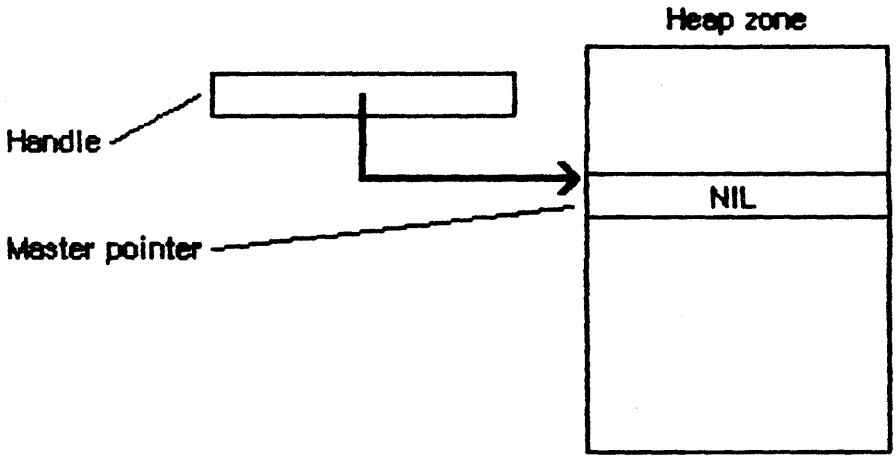
Next the Memory Manager tries to free space by purging blocks from the zone. Only relocatable blocks can be purged, and then only if they're explicitly marked as unlocked and purgeable. Purging a block removes it from its heap zone and frees the space it occupies. The block's master pointer is set to NIL, but the space occupied by the master pointer itself remains allocated. Any handles to the block now point to a NIL master pointer, and are said to be empty. If your program later needs to refer to the purged block, it can detect that the handle has become empty and ask the Memory Manager to reallocate the block. This operation updates the original master pointer, so that all handles to the block are left referring correctly to its new location (see Figure 7).

(eye)

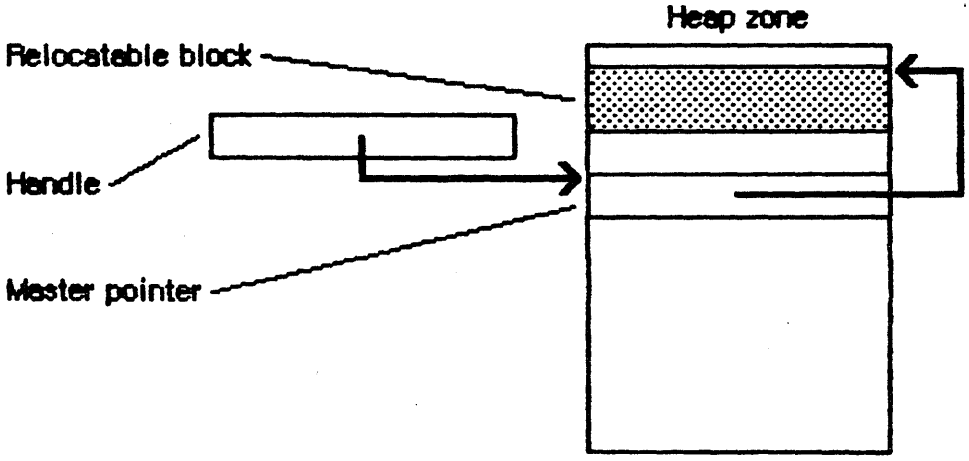
Reallocating a block only recovers the space it occupies, not its contents. Any information the block contains is lost when the block is purged. It's up to your program to reconstitute the block's contents after reallocating it.



Before purging



After purging



After reallocating

Figure 7. Purging and Reallocating a Block

Finally, if all else fails, the Memory Manager calls the grow zone function, if any, for the current heap zone. This is an optional routine that you can provide to take any last-ditch measures your program may have at its disposal to try to free some space in the zone. The term "grow zone function" is misleading, since the function doesn't actually attempt to "grow" (expand) the zone. Rather, its purpose is to try to create additional free space within the existing zone (such as by purging blocks that were previously marked unpurgeable) or reduce the fragmentation of existing free space (such as by unlocking previously locked blocks). The Memory Manager will call the grow zone function repeatedly, compacting the heap again after each call, until either it finds the space it's looking for or the grow zone function reports that it can offer no further help. In the latter case, the Memory Manager will give up and report that it's unable to satisfy your allocation request.

THE STACK AND THE HEAP

The application heap zone and the application stack share the same area in memory, growing toward each other from opposite ends (see Figure 8). Naturally it would be disastrous for either to grow so far that it collides with and overwrites the other. To help prevent such collisions, the Memory Manager enforces a limit on how far the application heap zone can grow toward the stack. Your program can set this application heap limit to control the allotment of available space betwen the stack and the heap.

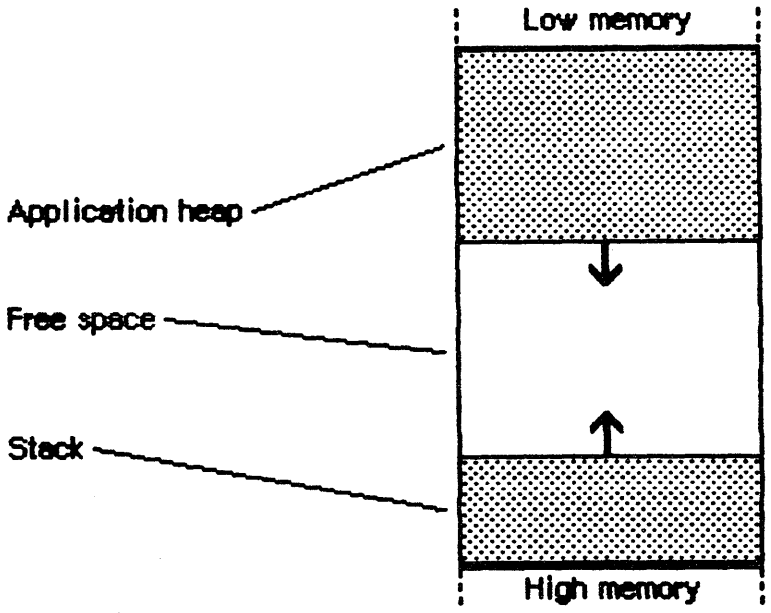


Figure 8. The Stack and the Heap

The application heap limit marks the boundary between the space available for the application heap zone and that reserved exclusively for the stack. At the start of each application program, the limit is initialized to allow 8K bytes for the stack. Depending on your program's needs, you can then adjust the limit to allow more heap space at the expense of the stack or vice versa.

Notice, however, that the limit applies only to expansion of the **heap**; it has no effect on how far the **stack** can expand. That is, although the heap can never expand beyond the limit into space reserved for the stack, there's nothing to prevent the stack from crossing the boundary and encroaching on space allotted for heap expansion--or even from overwriting part of the heap itself. It's up to you to set the limit low enough to allow for the maximum stack depth your program will ever need.

(hand)

> Regardless of the limit setting, the application zone is never allowed to grow to within 1K of the current end of the stack. This gives a little extra protection in case the stack is approaching the boundary or has crossed over onto the heap's side, and allows some safety margin for the stack to expand even further.

To help detect collisions between the stack and the heap, a "stack sniffer" routine is run sixty times a second, during the Macintosh's vertical retrace interrupt. This routine compares the current ends of the stack and the heap and opens an alert box on the screen in case of a collision. The stack sniffer can't prevent collisions, only detect them after the fact: a lot of computation can take place in a sixtieth of a second. In fact, the stack can easily expand into the heap, overwrite it, and then shrink back again before the next activation of the stack sniffer, escaping detection completely. The stack sniffer is useful mainly during software development; the alert box it displays can be confusing to your program's end user. Its purpose is to warn you, the programmer, that your program's stack and heap are colliding, so that you can adjust the heap limit to correct the problem before the user ever encounters it.

UTILITY DATA TYPES

The Memory Manager includes a number of type definitions for general-purpose use. For working with pointers and handles to allocated blocks, there are the following definitions:

```
TYPE SignedByte = -128..127;
     Byte       = Ø..255;
     Ptr        = ^SignedByte;
     Handle     = ^Ptr;
```

SignedByte stands for an arbitrary byte in memory, just to give Ptr and Handle something to point to. You can define a buffer of bufSize

untyped memory bytes as a PACKED ARRAY [1..bufSize] OF SignedByte. Byte is an alternative definition that treats byte-length data as unsigned rather that signed quantities.

Because of Pascal's strong typing rules, you can't directly assign a value of type Ptr to a variable of some other pointer type. Instead, you have to use the Lisa Pascal functions ORD and POINTER to convert the pointer to an integer address and then back to a pointer. For example, after the declarations

```
VAR aPtr:          Ptr;
    somethingElse: ^Thing;
```

you can make somethingElse point to the same object as aPtr with the assignment

```
somethingElse := POINTER(ORD(aPtr))
```

This works because POINTER returns a generalized "pointer to anything" (like the Pascal pointer constant NIL) that can be assigned to any variable of pointer type or supplied as an argument value for any routine parameter of pointer type.

Type ProcPtr, defined as

```
TYPE ProcPtr = Ptr;
```

is useful for treating procedures and functions as data objects. If aProcPtr is a variable of type ProcPtr and myProc is a procedure (or function) defined in your program, you can make aProcPtr point to myProc by using Lisa Pascal's @ operator:

```
aProcPtr := @myProc
```

Like the POINTER function, the @ operator produces a "pointer to anything". Using it, you can assign procedures and functions to variables of type ProcPtr, embed them in data structures, and pass them as arguments to other routines. Notice, however, that a ProcPtr technically points to a SignedByte, not an actual routine. As a result, there's no way in Pascal to access the underlying routine in order to call it. Only routines written in assembly language (such as those in the Operating System and the Toolbox) can actually call the routine designated by a ProcPtr.

For specifying the sizes of blocks on the heap, the Memory Manager defines a special type called Size:

```
TYPE Size = LongInt;
```

All Memory Manager routines that deal with block sizes expect parameters of type Size or return them as results. To specify a size bigger than any existing block, you can use the constant maxSize:

```
CONST maxSize = $800000;
```

This is an enormous value, equivalent to 8 megabytes or 8,388,608 bytes --more than forty times the Macintosh's total memory capacity!

MEMORY MANAGER DATA STRUCTURES

This section contains detailed information on the Memory Manager's internal data structures. You won't need this information if you're just using the Memory Manager routinely to allocate and release blocks of memory from the application heap zone. The details are included here for programmers with unusual needs (or who are just curious about how the Memory Manager works).

Structure of Heap Zones

Each heap zone begins with a 52-byte zone header and ends with a 12-byte zone trailer (see Figure 9). The header contains all the information the Memory Manager needs about that heap zone; the trailer is just a minimum-size free block (described in the next section) placed at the end of the zone as a marker. All the remaining space between the header and trailer is available for allocation.

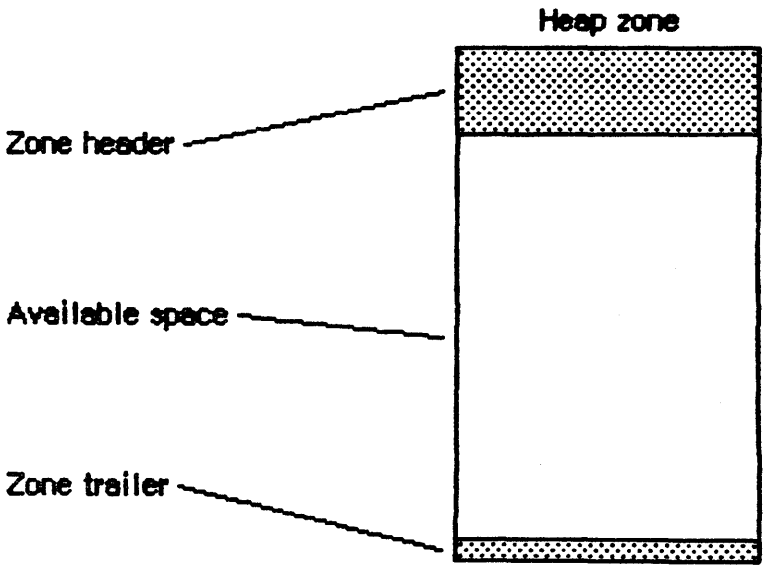


Figure 9. Structure of a Heap Zone

In Pascal, a heap zone is defined as a zone record of type Zone, reflecting the structure of the zone header. It's always referred to with a zone pointer of type THz ("the heap zone"):

```
TYPE THz  = ^Zone;
     Zone = RECORD
               bkLim:      Ptr;
               purgePtr:   Ptr;
               hFstFree:   Ptr;
               zcbFree:    LongInt;
               gzProc:     ProcPtr;
               moreMast:   INTEGER;
               flags:      INTEGER;
               cntRel:     INTEGER;
               maxRel:     INTEGER;
               cntNRel:    INTEGER;
               maxNRel:    INTEGER;
               cntEmpty:   INTEGER;
               cntHandles: INTEGER;
               minCBFree:  LongInt;
               purgeProc:  ProcPtr;
               sparePtr:   Ptr;
               allocPtr:   Ptr;
               heapData:   INTEGER
            END;
```

(eye)

The fields of the zone header are for the Memory Manager's own internal use. You can examine the contents of the zone's fields, but in general it doesn't make sense for your program to try to change them. The few exceptions are noted below in the discussions of the specific fields.

BkLim is a pointer to the zone's trailer block. Since the trailer is the last block in the zone, this constitutes a limit pointer to the memory byte **following** the last byte of usable space in the zone.

PurgePtr and allocPtr are "roving pointers" into the heap zone that the Memory Manager maintains for its own internal use. When scanning the zone for a free block to satisfy an allocation request, the Memory Manager begins at the block pointed to by allocPtr instead of always starting from the beginning of the zone. When purging blocks from the zone, it starts from the block pointed to by purgePtr.

HFstFree is a pointer to the first free master pointer in the zone. Instead of just allocating space for one master pointer each time a relocatable block is created, the Memory Manager "preallocates" several master pointers at a time, themselves forming a nonrelocatable block within the zone. The moreMast field of the zone record tells the Memory Manager how many master pointers at a time to preallocate for this zone. Master pointers for the system heap zone are allocated 32 at a time; for the application zone, 64 at a time. For other heap zones, you specify the value of moreMast when you create the zone.

All master pointers that are allocated but not currently in use are linked together into a list beginning in the hFstFree field. When you

allocate a new relocatable block, the Memory Manager removes the first available master pointer from this list, sets it to point to the new block, and returns its address to you as a handle to the block. (If the list is empty, it allocates a fresh block of moreMast master pointers, uses one of them for the new relocatable block, and adds the rest to the list.) When you release a relocatable block, its master pointer isn't released, but linked onto the beginning of the list to be reused. Thus the amount of space devoted to master pointers can increase, but can never decrease unless the zone is reinitialized (for example, at the start of a new application program).

The zcbFree field always contains the number of free bytes remaining in the zone ("zcb" stands for "zone count of bytes"). As blocks are allocated and released, the Memory Manager adjusts zcbFree accordingly. This number represents an upper limit on the size of block you can allocate from this heap zone.

(eye)

> It may not actually be possible to allocate a block as big as zcbFree bytes. As space in a heap zone becomes fragmented, the free bytes typically don't remain contiguous but become scattered throughout the zone. Because nonrelocatable and locked blocks can't be moved, it isn't always possible to collect all the free space into a single block by compaction. (Even if the zone contains only relocatable blocks, the master pointers to these blocks are themselves nonrelocatable "islands" that can interfere with the compaction process.) So the maximum-size block you can actually allocate from the zone may be appreciably smaller than zcbFree bytes.

The gzProc field is a pointer to the zone's grow zone function, or NIL if there is none. You supply this pointer when you create a new heap zone and can change it at any time with the SetGrowZone procedure. The system and application heap zones initially have no grow zone function.

Flags contains a set of flag bits strictly for the Memory Manager's internal use; your program should never need to access this field.

CntRel, maxRel, cntNRel, maxNRel, cntEmpty, cntHandles, and minCBFree are not used by the ROM-based version of the Memory Manager. *** These fields are reserved for eventual use by a special RAM-based version that will gather statistics on a program's memory usage within each heap zone. CntRel and cntNRel will be used to count, respectively, the number of relocatable and nonrelocatable blocks currently allocated within the zone. MaxRel and maxNRel will record the "historical maximum" values attained by cntRel and cntNRel since the program was started. CntEmpty will count the current number of empty master pointers, cntHandles the total number of master pointers currently allocated. MinCBFree will record the historical minimum number of free bytes in the zone. ***

PurgeProc is a pointer to the zone's purge warning procedure (sometimes called a "purge hook"), or NIL if there is none. The Memory Manager

will call this procedure whenever it purges a block from the zone. You can "install" a purge warning procedure in this field to do optional housekeeping such as writing out a block's contents to a disk file before it's purged. In fact, this is exactly the way the Resource Manager keeps the contents of resources up to date if they're changed by your program. If you want to install your own purge hook, you have to be very careful not to interfere with the one the Resource Manager may have installed; see "Special Techniques", later in this manual, for further details.

SparePtr is an extra field included in the zone header for possible future expansion.

The last field of a zone record, heapData, is a dummy field marking the beginning of the zone's usable memory space. HeapData nominally contains an integer, but this integer has no significance in itself--it's just the first two bytes in the block header of the first block in the zone. The purpose of the heapData field is to give you a way of locating the effective beginning of the zone. For example, if myZone is a zone pointer, then

```
@(myZone^.heapData)
```

is a pointer to the first usable byte in the zone, just as

```
myZone^.bkLim
```

is a limit pointer to the byte following the last usable byte in the zone.

Structure of Blocks

Every memory block in a heap zone, whether allocated or free, has a block header that the Memory Manager uses to find its way around in the zone. Block headers are completely transparent to your program. All pointers and handles to allocated blocks point to the beginning of the block's contents, following the end of the header. Similarly, all block sizes seen by your program refer to the block's logical size (the number of bytes in its contents) rather than its physical size (the number of bytes it actually occupies in memory, including the header and any unused bytes at the end of the block).

Since your program shouldn't normally have to deal with block headers directly, there's no Pascal record type defining their structure. (It's possible to access block headers in assembly language, but be sure you know what you're doing!) A block header consists of 8 bytes, as shown in Figure 1Ø.

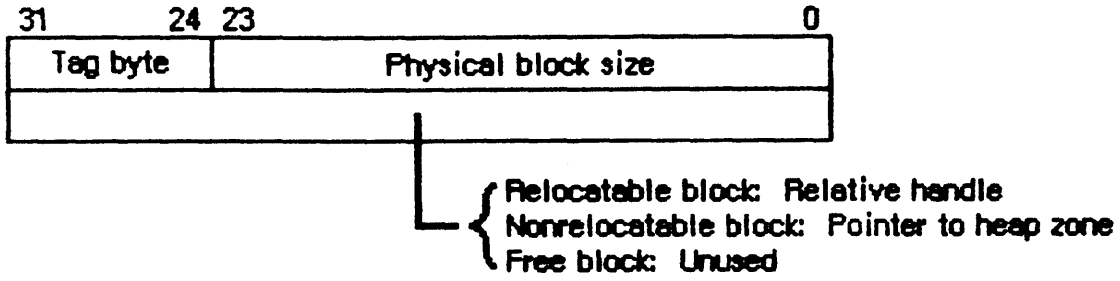


Figure 1Ø. Block Header

The first byte of the block header is the tag byte, discussed in detail below. The next 3 bytes contain the block's physical size in bytes. Adding this number to the block's address gives the address of the next block in the zone.

The contents of the second long word (4 bytes) in the block header depend on the type of block. For relocatable blocks, it contains the block's relative handle: a pointer to the block's master pointer, expressed as an offset relative to the start of the heap zone rather than as an absolute memory address. Adding the relative handle to the zone pointer produces a true handle for this block. For nonrelocatable blocks, the second long word of the header is just a pointer to the block's zone. For free blocks, these 4 bytes are unused.

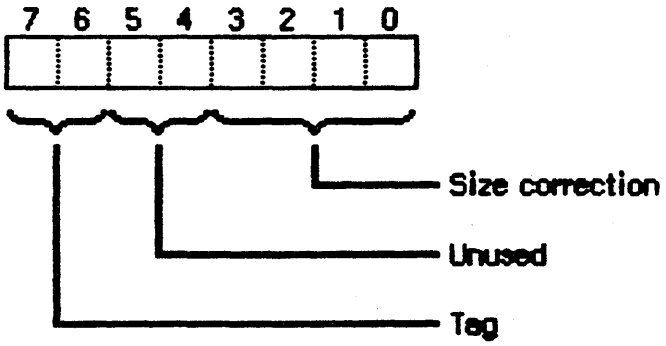


Figure 11. Tag Byte

The tag byte consists of a 2-bit tag, 2 unused bits, and a 4-bit size correction, as shown in Figure 11. The tag identifies the type of block:

Tag	Block type
ØØ	Free
Ø1	Nonrelocatable
1Ø	Relocatable

(A tag value of 11 is invalid.)

The size correction is the number of unused bytes at the end of the block, beyond the end of the block's contents. It's equal to the difference between the block's logical and physical sizes, excluding the 8 bytes of overhead for the block header:

```
sizeCorrection = physicalSize - logicalSize - 8
```

There are several reasons why a block may contain such unused bytes:

- The Memory Manager allocates space only in whole 16-bit words--that is, in even numbers of bytes. If the block's logical size is odd, an extra, unused byte is added at the end to keep the physical size even.

- Earlier versions of the Memory Manager used a block header of 12 bytes instead of 8. Although the header is now only 8 bytes long, the Memory Manager still enforces a minimum size of 12 bytes per block for compatibility with these earlier versions. If the logical size of a block is less than 4, enough extra bytes are allocated at the end of the block to bring its physical size up to 12.

- The 12-byte minimum applies to all blocks, free as well as allocated. If allocating the required number of bytes from a free block would leave a fragment of fewer than 12 free bytes, the leftover bytes are included unused at the end of the newly allocated block instead of being returned to free storage.

Putting all this together, the minimum overhead required for each allocated block is 8 bytes for the block header, plus an additional 4 bytes for the master pointer if the block is relocatable. The maximum possible overhead is 26 bytes, for a relocatable block with a logical size of Ø being allocated from a free block of 22 bytes: 8 bytes for the header, 4 for the master pointer, 4 to satisfy the 12-byte minimum, and a leftover fragment of 1Ø free bytes that's too small to return to free storage.

Structure of Master Pointers

The master pointer to a relocatable block has the structure shown in Figure 12. The low-order 3 bytes of the long word contain the address of the block's contents. The high-order byte contains some flag bits that specify the block's current status. Bit 7 of this byte is the lock bit (1 if the block is locked, Ø if it's unlocked); bit 6 is the purge bit (1 if the block is purgeable, Ø if it's unpurgeable). Bit 5

is used by the Resource Manager to identify blocks containing resource information for special treatment; such resource blocks are marked by a 1 in this bit.

(eye)

Before attempting to compare one master pointer with another or perform any arithmetic operation on it, don't forget to strip off the flag bits in the high-order byte.

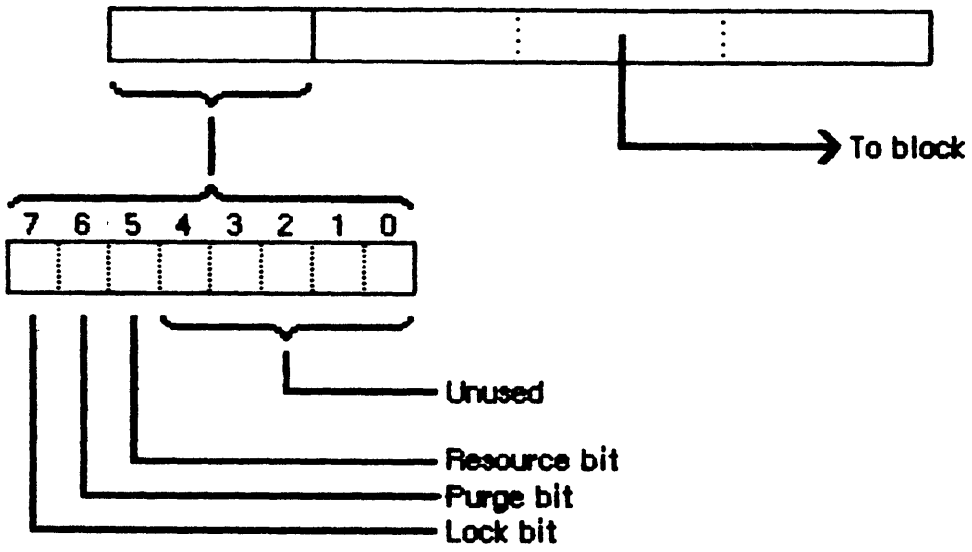


Figure 12. Structure of a Master Pointer

RESULT CODES

Like most other Operating System routines, Memory Manager routines generally return a result code in addition to their normal results. This is an integer code indicating whether the routine completed its task successfully or was prevented by some error condition. The type definition for result codes is

```
TYPE MemErr = INTEGER;
```

In the normal case that no error is detected, the result code is Ø; a nonzero result code signals an error:

```
CONST noErr         = Ø;      {no error}
      memFullErr    = -1Ø8;   {not enough room in zone}
      nilHandleErr  = -1Ø9;   {NIL master pointer}
      memWZErr      = -111;   {attempt to operate on a free block}
      memPurErr     = -112;   {attempt to purge a locked block}
```

To inspect a result code from Pascal, call the Memory Manager function MemError. This function always returns the result code from the last Memory Manager call.

Assembly-language note: When called from assembly language via the trap mechanism, not all Memory Manager routines return a result code. Those that do always leave it as a word-length quantity in the low-order half of register DØ on return from the trap. However, some routines leave something else there instead: see the descriptions of individual routines for details. Just before returning, the trap dispatcher tests the lower half of DØ with a TST.W instruction, so that on return from the trap the condition codes reflect the status of the result code, if any.

The stack-based interface routines called from Pascal always produce a result code. If the underlying trap doesn't return one, the interface routine "manufactures" a result code of noErr and stores it where it can later be accessed with MemError.

The ROM-based version of the Memory Manager does only limited error checking. This manual describes only the result codes reported by the ROM version. *** There may eventually be a special RAM-based version that will do more extensive error checking. If so, any additional result codes reported by the RAM version will be documented at that time. ***

USING THE MEMORY MANAGER

This section discusses how the Memory Manager routines fit into the general flow of your program and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in the next section.

Assembly-language note: If you're writing code that will be executed via a hardware interrupt, you can't use the Memory Manager. This is because an interrupt can occur unpredictably at any time. In particular, it can occur while the Memory Manager is in the middle of a heap compaction or in some other inconsistent internal state. To prevent catastrophes, interrupt routines are not allowed to allocate space from the heap.

There's ordinarily no need to initialize the Memory Manager before using it. The system heap zone is automatically initialized each time

the system is started up, and the application heap zone each time an application program is launched. In the unlikely event that you need to reinitialize the application zone while your program is running, you can use InitApplZone.

You can create additional heap zones for your program's own use, either from within the original application zone or from the stack, with InitZone. If you do maintain more than one heap zone, you can find out which zone is current at any given time with GetZone and switch from one to another with SetZone. Almost all Memory Manager operations implicitly apply to the current heap zone. To refer to the system heap zone or the (original) application heap zone, use the Memory Manager function SystemZone or ApplicZone. To find out which zone a particular block resides in, use HandleZone (if the block is relocatable) or PtrZone (if it's nonrelocatable).

(hand)

Most applications will just use the original application heap zone and never have to worry about which zone is current.

The main work of the Memory Manager is allocating and releasing blocks of memory. To allocate a new relocatable block, use NewHandle; for a nonrelocatable block, use NewPtr. These functions return a handle or a pointer, as the case may be, to the newly allocated block. You then use that handle or pointer whenever you need to refer to the block.

To release a block when you're finished with it, use DisposHandle or DisposPtr. You can also change the size of an already allocated block with SetHandleSize or SetPtrSize, and find out its current size with GetHandleSize or GetPtrSize. Use HLock and HUnlock to lock and unlock relocatable blocks.

(hand)

In general, you should use relocatable blocks whenever possible, to avoid unnecessary fragmentation of free space. Use nonrelocatable blocks only for things like I/O buffers, queues, and other objects that must have a fixed location in memory. For most applications, the only Memory Manager routines you'll ever need will be NewHandle, DisposHandle, and SetHandleSize.

(hand)

If you must lock a relocatable block, try to unlock it again at the earliest possible opportunity. Before allocating a block that you know will be locked for long periods of time, call ReservMem to make room for the block as near as possible to the beginning of the zone.

To speed up your program, you may sometimes want to convert the handle to a relocatable block into a copy of the master pointer it points to. This is called dereferencing the handle, and allows you to refer to the block by single instead of double indirection. Dereferencing a handle can be dangerous if you aren't careful; see "Special Techniques" for

further information. If you ever need to convert a dereferenced master pointer back into the original handle, use RecoverHandle.

Ordinarily, you shouldn't have to worry about compacting the heap or purging blocks from it; the Memory Manager automatically takes care of these chores for you. You can control which blocks are purgeable with HPurge and HNoPurge. If for some reason you want to compact or purge the heap explicitly, you can do so with CompactMem or PurgeMem. To explicitly purge a specific block, use EmptyHandle.

(eye)

> If you're working with purgeable blocks, **be careful!** Such blocks may be removed from the heap zone at any time in order to satisfy a memory allocation request. So before attempting to access any purgeable block, always check its handle to make sure the block is still allocated. If the handle is empty (that is, if h^ = NIL, where h is the handle), then the block has been purged: before accessing it, you have to reallocate it and update its master pointer by calling ReallocHandle. (If it's a resource block, use the Resource Manager procedure LoadResource instead.)

You can find out how much free space is left in a heap zone by calling FreeMem (to get the total number of free bytes) or MaxMem (to get the size of the largest single free block and the maximum amount by which the zone can grow). Beware, however: MaxMem also compacts and purges the entire zone before returning this information. To limit the growth of the application zone, use SetApplLimit; to install a grow zone function to help the Memory Manager allocate space in a zone, use SetGrowZone.

After calling any Memory Manager routine, you can examine its result code with MemError.

MEMORY MANAGER ROUTINES

This section describes all the Memory Manager procedures and functions. Each routine is presented first in its Pascal form (if there is one). For most routines, this is followed by a box containing information needed to use the routine from assembly language. Most Pascal programmers can just skip this box, although the list of result codes may be of interest to some. For general information on using the Memory Manager from assembly language, see "Using the Operating System from Assembly Language" *** (to be written) *** and also "Notes for Assembly-Language Programmers" in this manual.

Initialization and Allocation

PROCEDURE InitApplZone;

Trap macro	_InitApplZone		
On exit	D0:	result code (integer)	
Result codes	0	$0000 noErr	No error

InitApplZone initializes the application heap zone and makes it the current zone. The contents of any previous application zone are completely wiped out; all previously existing blocks in that zone are discarded. InitApplZone is called by the Segment Loader when launching an application program; you shouldn't normally need to call it from within your own program.

(eye)

> Reinitializing the application zone from within a running program is tricky, since the program's code itself resides in the application zone. To do it safely, you have to move the code of the running program into the **system** heap zone, jump to it there, reinitialize the application zone, move the code back into the application zone, and jump to it again. Don't attempt this operation unless you're sure you know what you're doing.

The application zone has a standard initial size of 6K bytes, immediately following the end of the system heap zone, and can be expanded as needed in 1K increments. Space is initially allocated for 64 master pointers; should more be needed later, they will be added 64 at a time. The zone's grow zone function is set to NIL. After a call to InitApplZone, MemError will always return noErr.

```
PROCEDURE SetApplBase (startPtr: Ptr);
```

Trap macro	_SetApplBase	
On entry	AØ:	startPtr (pointer)
On exit	DØ:	result code (integer)
Result codes	Ø $ØØØØ	noErr No error

SetApplBase changes the starting address of the application heap zone to the address designated by startPtr, reinitializes the zone, and makes it the current zone. The contents of any previous application zone are completely wiped out; all previously existing blocks in that zone are discarded. SetApplBase is normally called only by the system itself; you should never need to call this procedure from within your own program.

Since the application heap zone begins immediately following the end of the system zone, changing its starting address has the effect of changing the size of the system zone. The system zone can be made larger, but never smaller; if startPtr points to an address lower than the current end of the system zone, it's ignored and the application zone's starting address is left unchanged.

In any case, SetApplBase reinitializes the application zone to its standard initial size of 6K bytes, which can later be expanded as needed in 1K increments. Space is initially allocated for 64 master pointers; should more be needed later, they will be added 64 at a time. The zone's grow zone function is set to NIL. After a call to SetApplBase, MemError will always return noErr.

(eye)

> Like InitApplZone, SetApplBase is a tricky operation, because the code of the program itself resides in the application heap zone. The recommended procedure for doing it safely is the same as for InitApplZone (see above); again, don't attempt it unless you know what you're doing.

```
PROCEDURE InitZone (growProc: ProcPtr; masterCount: INTEGER; limitPtr,
          startPtr: Ptr);
```

Trap macro _InitZone

On entry AØ: pointer to parameter block

startPtr	(4-byte pointer)
limitPtr	(4-byte pointer)
masterCount	(2-byte integer)
growProc	(4-byte pointer)

On exit DØ: result code (integer)

Result codes				
	Ø	$ØØØØ	noErr	No error

InitZone creates a new heap zone, initializes its header and trailer, and makes it the current zone. The startPtr parameter is a pointer to the first byte of the new zone; limitPtr points to the byte **following** the end of the zone. That is, the new zone will occupy memory addresses from ORD(startPtr) to ORD(limitPtr) - 1.

MasterCount tells how many master pointers should be allocated at a time for the new zone. The specified number of master pointers are created initially; should more be needed later, they will be added in increments of this same number. For the system heap zone, masterCount is 32; for the application heap zone, it's 64.

The growProc parameter is a pointer to the grow zone function for the new zone, if any. If you're not defining a grow zone function for this one, supply a NIL value for growProc.

The new zone includes a 52-byte header and a 12-byte trailer, so its actual usable space runs from ORD(startPtr) + 52 through ORD(limitPtr) - 13. In addition, each master pointer occupies 4 bytes within this usable area. Thus the total available space in the zone, in bytes, is initially

```
ORD(limitPtr) - ORD(startPtr) - 64 - 4*masterCount
```

This number must not be less than Ø. Note that the amount of available space in the zone may decrease as more master pointers are allocated.

After a call to InitZone, MemError will always return noErr.

PROCEDURE SetApplLimit (zoneLimit: Ptr);

Trap macro	_SetApplLimit	
On entry	A0:	zoneLimit (pointer)
On exit	D0:	result code (integer)
Result codes	0 $0000	noErr No error

SetApplLimit sets the application heap limit, beyond which the application heap zone can't be expanded. The actual expansion isn't under your program's control, but is done automatically by the Memory Manager when necessary in order to satisfy an allocation request. Only the original application zone can be expanded.

ZoneLimit is a limit pointer to a byte in memory beyond which the zone will not be allowed to grow. That is, the zone can grow to include the byte **preceding** zoneLimit in memory, but no farther. If the zone already extends beyond the specified limit it won't be cut back, but it will be prevented from growing any more.

(eye)

Notice that zoneLimit is **not** a byte count. To limit the application zone to a particular size (say 8K bytes), you have to write something like

```
SetApplLimit(POINTER(ORD(ApplicZone) + 8192))
```

After a call to SetApplLimit, MemError will always return noErr.

Heap Zone Access

FUNCTION GetZone : THz;

Trap macro	_GetZone	
On exit	A0:	function result (pointer)
	0:	result code (integer)
Result codes	0	$0000 noErr No error

GetZone returns a pointer to the current heap zone. After the call, MemError will always return noErr.

PROCEDURE SetZone (hz: THz);

Trap macro	_SetZone	
On entry	A0:	hz (pointer)
On exit	D0:	result code (integer)
Result codes	0	$0000 noErr No error

SetZone sets the current heap zone to the zone pointed to by hz. After the call, MemError will always return noErr.

FUNCTION SystemZone : THz; [Pascal only]

Trap macro	None	
Result codes	0	$0000 noErr No error

SystemZone returns a pointer to the system heap zone. After the call, MemError will always return noErr.

Assembly-language note: SystemZone is part of the Pascal interface to the Memory Manager, not part of the Memory Manager

itself. It doesn't reside in ROM and can't be called via a trap. To get a pointer to the system heap zone from assembly language, use the global variable sysZone.

FUNCTION ApplicZone : THz; [Pascal only]

Trap macro None

Result codes Ø $ØØØØ noErr No error

ApplicZone returns a pointer to the original application heap zone. After the call, MemError will always return noErr.

Assembly-language note: ApplicZone is part of the Pascal interface to the Memory Manager, not part of the Memory Manager itself. It doesn't reside in ROM and can't be called via a trap. To get a pointer to the application heap zone from assembly language, use the global variable applZone.

Allocating and Releasing Relocatable Blocks

FUNCTION NewHandle (logicalSize: Size) : Handle;

Trap macro _NewHandle

On entry DØ: logicalSize (long integer)

On exit AØ: function result (handle)
Ø: result code (integer)

Result codes			
Ø	$ØØØØ	noErr	No error
-1Ø8	$FF94	memFullErr	Not enough room in zone

NewHandle allocates a new relocatable block from the current heap zone and returns a handle to it (or NIL if a block of that size can't be created). The new block will have a logical size of logicalSize bytes and will initially be marked unlocked and unpurgeable.

NewHandle will pursue all avenues open to it in order to create a free block of the requested size, including compacting the heap zone, increasing its size, purging blocks from it, and calling its grow zone function, if any. If all such attempts fail, or if the zone has run out of free master pointers and there's no room to allocate more, NewHandle returns NIL and MemError will return memFullErr after the call. If a new block was successfully allocated, NewHandle returns a handle to the new block and MemError will return noErr.

```
PROCEDURE DisposHandle (h: Handle);
```

Trap macro	_DisposHandle			
On entry	A0:	h (handle)		
On exit	A0:	0		
	D0:	result code (integer)		
Result codes	0	$0000	noErr	No error
	-111	$FF91	memWZErr	Attempt to operate on a free block

DisposHandle releases the space occupied by the relocatable block whose handle is h. If the block is already free, MemError will return memWZErr after the call; otherwise it will return noErr.

(eye)
After a call to DisposHandle, all handles to the released block become invalid and should not be used again.

```
FUNCTION GetHandleSize (h: Handle) : Size;
```

Trap macro	_GetHandleSize			
On entry	A0:	h (handle)		
On exit	D0:	if >= 0, function result (long integer)		
		if < 0, result code (integer)		
Result codes	0	$0000	noErr	No error [Pascal only]
	-109	$FF93	nilHandleErr	NIL master pointer
	-111	$FF91	memWZErr	Attempt to operate on a free block

GetHandleSize returns the logical size, in bytes, of the relocatable block whose handle is h. After the call, MemError will return

nilHandleErr if h points to a NIL master pointer, memWZErr if h is the handle of a free block, and noErr otherwise. In case of an error, GetHandleSize returns a result of Ø.

Assembly-language note: Recall that the trap dispatcher sets the condition codes before returning from a trap by testing the low-order half of register DØ with a TST.W instruction. Since the block size returned in DØ by _GetHandleSize is a full 32-bit long word, the word-length test sets the condition codes incorrectly in this case. To branch on the contents of DØ, use your own TST.L instruction on return from the trap to test the full 32 bits of the register.

```
PROCEDURE SetHandleSize (h: Handle; newSize: Size);
```

Trap macro	_SetHandleSize			
On entry	AØ:	h (handle)		
	DØ:	newSize (long integer)		
On exit	DØ:	result code (integer)		
Result codes	Ø	$ØØØØ	noErr	No error
	-1Ø8	$FF94	memFullErr	Not enough room to grow
	-1Ø9	$FF93	nilHandleErr	NIL master pointer
	-111	$FF91	memWZErr	Attempt to operate on a free block

SetHandleSize changes the logical size of the relocatable block whose handle is h to newSize bytes. After the call, MemError will return memFullErr if newSize is greater than the block's current size and enough room can't be found for the block to grow, nilHandleErr if h points to a NIL master pointer, memWZErr if h is the handle of a free block, and noErr otherwise.

```
FUNCTION HandleZone (h: Handle) : THz;
```

Trap macro	_HandleZone			
On entry	A0:	h (handle)		
On exit	A0:	function result (pointer)		
	D0:	result code (integer)		
Result codes	0	$0000	noErr	No error
	-111	$FF91	memWZErr	Attempt to operate on a free block

HandleZone returns a pointer to the heap zone containing the relocatable block whose handle is h.

If handle h is empty (points to a NIL master pointer), HandleZone returns a pointer to the current heap zone and doesn't report an error: after the call, MemError will return noErr. If h is the handle of a free block, MemError will return memWZErr; in this case, the result returned by HandleZone is meaningless and should be ignored.

```
FUNCTION RecoverHandle (p: Ptr) : Handle;
```

Trap macro	_RecoverHandle			
On entry	A0:	p (pointer)		
On exit	A0:	function result (handle)		
	D0:	unchanged (!)		
Result codes	0	$0000	noErr	No error [Pascal only]

RecoverHandle returns a handle to the relocatable block pointed to by p. If you've "dereferenced" a handle (converted it to a simple pointer) for efficiency, you can use this function to get back the original handle. After the call, MemError will always return noErr.

Assembly-language note: Through a minor oversight, the trap _RecoverHandle neglects to return a result code in register D0; the previous contents of D0 are preserved unchanged. The stack-based interface routine called from Pascal always produces a result code of noErr.

```
PROCEDURE ReallocHandle (h: Handle; logicalSize: Size);
```

Trap macro	_ReallocHandle			
On entry	A0:	h (handle)		
	D0:	logicalSize (long integer)		
On exit	A0:	original h or NIL		
	D0:	result code (integer)		
Result codes	0	$0000	noErr	No error
	-108	$FF94	memFullErr	Not enough room in zone
	-111	$FF91	memWZErr	Attempt to operate on a free block
	-112	$FF90	memPurErr	Block is locked

ReallocHandle allocates a new relocatable block with a logical size of logicalSize bytes. It then updates handle h by setting its master pointer to point to the new block. The main use of this procedure is to reallocate space for a block that has been purged. Normally h is an empty handle, but it need not be: if it points to an existing block, that block is released before the new block is created.

After the call, MemError will return noErr if ReallocHandle succeeds in allocating a block of the requested size; if room can't be made for the requested block, it will return memFullErr. If h is the handle of an existing block, MemError will return memPurErr if the block is locked and memWZErr if it's already free. In case of an error, no new block is allocated and handle h is left unchanged.

Assembly-language note: On return from _ReallocHandle, register A0 contains the original handle h, or 0 (NIL) if no room could be found for the requested block.

Allocating and Releasing Nonrelocatable Blocks

```
FUNCTION NewPtr (logicalSize: Size) : Ptr;
```

Trap macro	_NewPtr			
On entry	D0:	logicalSize (long integer)		
On exit	A0:	function result (pointer)		
	D0:	result code (integer)		
Result codes	0	$0000	noErr	No error
	-108	$FF94	memFullErr	Not enough room in zone

NewPtr allocates a new nonrelocatable block from the current heap zone and returns a pointer to it (or NIL if a block of that size can't be created). The new block will have a logical size of logicalSize bytes.

NewPtr will pursue all avenues open to it in order to create a free block of the requested size, including compacting the heap zone, increasing its size, purging blocks from it, and calling its grow zone function, if any. If all such attempts fail, NewPtr returns NIL and MemError will return memFullErr after the call. If a new block was successfully allocated, NewPtr returns a pointer to the new block and MemError will return noErr.

```
PROCEDURE DisposPtr (p: Ptr);
```

Trap macro	_DisposPtr			
On entry	A0:	p (pointer)		
On exit	A0:	0		
	D0:	result code (integer)		
Result codes	0	$0000	noErr	No error
	-111	$FF91	memWZErr	Attempt to operate on a free block

DisposPtr releases the space occupied by the nonrelocatable block pointed to by p. If the block is already free, MemError will return memWZErr after the call; otherwise it will return noErr.

(eye)
After a call to DisposPtr, all pointers to the released block become invalid and should not be used again.

FUNCTION GetPtrSize (p: Ptr) : Size;

Trap macro	_GetPtrSize			
On entry	A0:	p (pointer)		
On exit	D0:	if >= 0, function result (long integer) if < 0, result code (integer)		
Result codes	0	$0000	noErr	No error [Pascal only]
	-111	$FF91	memWZErr	Attempt to operate on a free block

GetPtrSize returns the logical size, in bytes, of the nonrelocatable block pointed to by p. After the call, MemError will return memWZErr if p points to a free block and noErr otherwise. In case of an error, GetPtrSize returns a result of 0.

Assembly-language note: Recall that the trap dispatcher sets the condition codes before returning from a trap by testing the low-order half of register D0 with a TST.W instruction. Since the block size returned in D0 by _GetPtrSize is a full 32-bit long word, the word-length test sets the condition codes incorrectly in this case. To branch on the contents of D0, use your own TST.L instruction on return from the trap to test the full 32 bits of the register.

```
PROCEDURE SetPtrSize (p: Ptr; newSize: Size);
```

Trap macro	_SetPtrSize			
On entry	A0:	p (pointer)		
	D0:	newSize (long integer)		
On exit	D0:	result code (integer)		
Result codes	0	$0000	noErr	No error
	-108	$FF94	memFullErr	Not enough room to grow
	-111	$FF91	memWZErr	Attempt to operate on a free block

SetPtrSize changes the logical size of the nonrelocatable block pointed to by p to newSize bytes. After the call, MemError will return memFullErr if newSize is greater than the block's current size and enough room can't be found for the block to grow, memWZErr if p points to a free block, and noErr otherwise.

```
FUNCTION PtrZone (p: Ptr) : THz;
```

Trap macro	_PtrZone			
On entry	A0:	p (pointer)		
On exit	A0:	function result (pointer)		
	D0:	result code (integer)		
Result codes	0	$0000	noErr	No error
	-111	$FF91	memWZErr	Attempt to operate on a free block

PtrZone returns a pointer to the heap zone containing the nonrelocatable block pointed to by p. If p points to a free block, MemError will return memWZErr after the call; in this case, the result returned by PtrZone is meaningless and should be ignored.

Freeing Space on the Heap

```
FUNCTION FreeMem : LongInt;
```

Trap macro	_FreeMem			
On exit	DØ:	function result (long integer)		
Result codes	Ø	$ØØØØ	noErr	No error [Pascal only]

FreeMem returns the total amount of free space in the current heap zone, in bytes. Notice that it may not actually be possible to allocate a block of this size, because of fragmentation due to nonrelocatable or locked blocks. After a call to FreeMem, MemError will always return noErr.

```
FUNCTION MaxMem (VAR grow: Size) : Size;
```

Trap macro	_MaxMem			
On exit	DØ:	function result (long integer)		
	AØ:	grow (long integer)		
Result codes	Ø	$ØØØØ	noErr	No error [Pascal only]

MaxMem compacts the current heap zone and purges all purgeable blocks from the zone. It returns as its result the size in bytes of the largest contiguous free block in the zone after the compaction. If the current zone is the original application heap zone, the variable parameter grow is set to the maximum number of bytes by which the zone can grow. For any other heap zone, grow is set to Ø. MaxMem doesn't actually expand the zone or call its grow zone function. After the call, MemError will always return noErr.

```
FUNCTION CompactMem (cbNeeded: Size) : Size;
```

Trap macro	_CompactMem			
On entry	D0:	cbNeeded (long integer)		
On exit	D0:	function result (long integer)		
	A0:	pointer to desired block or NIL		
Result codes	0	$0000	noErr	No error [Pascal only]

CompactMem compacts the current heap zone by moving relocatable blocks forward and collecting free space together until a contiguous block of at least cbNeeded free bytes is found or the entire zone is compacted. For each block that's moved, the master pointer is updated so that all handles to the block remain valid. CompactMem returns the size in bytes of the largest contiguous free block it finds, but doesn't actually allocate the block. After the call, MemError will always return noErr.

(hand)
To force a compaction of the entire heap zone, set cbNeeded equal to maxSize.

Assembly-language note: On return from _CompactMem, register A0 contains a pointer to a free block of at least cbNeeded bytes, or 0 (NIL) if no such block could be found.

```
FUNCTION ResrvMem (cbNeeded: Size);
```

Trap macro	_ResrvMem			
On entry	D0:	cbNeeded (long integer)		
On exit	A0:	pointer to desired block or NIL		
	D0:	result code (integer)		
Result codes	0	$0000	noErr	No error
	-108	$FF94	memFullErr	Not enough room in zone

ResrvMem creates free space for a block of cbNeeded contiguous bytes at the lowest possible position in the current heap zone. It will try every available means to place the block as close as possible to the

beginning of the zone, including moving other blocks upward, expanding the zone, or purging blocks from it. If a free block of at least the requested size can't be created, MemError will return memFullErr after the call; otherwise it will return noErr. Notice that ResrvMem doesn't actually allocate the block.

(hand)
When you allocate a relocatable block that you know will be locked for long periods of time, call ResrvMem first. This reserves space for the block near the beginning of the heap zone, where it will interfere with compaction as little as possible. It isn't necessary to call ResrvMem for a nonrelocatable block; NewPtr calls it automatically.

Assembly-language note: On return from _ResrvMem, register A0 contains a pointer to the desired free block of at least cbNeeded bytes, or 0 (NIL) if no such block could be created.

FUNCTION PurgeMem (cbNeeded: Size);

Trap macro	_PurgeMem			
On entry	D0:	cbNeeded (long integer)		
On exit	A0:	pointer to desired block or NIL		
	D0:	result code (integer)		
Result codes	0	$0000	noErr	No error
	-108	$FF94	memFullErr	Not enough room in zone

PurgeMem purges blocks from the current heap zone until a contiguous block of at least cbNeeded free bytes is created or the entire zone is purged. Only relocatable, unlocked, purgeable blocks can be purged. If a free block of at least the requested size is found, MemError will return noErr after the call; if not, it will return memFullErr. Notice that PurgeMem doesn't actually allocate the block.

(hand)
To force a purge of the entire heap zone, set cbNeeded equal to maxSize.

Assembly-language note: On return from _PurgeMem, register AØ contains a pointer to a free block of at least cbNeeded bytes, or Ø (NIL) if no such block could be found.

```
PROCEDURE EmptyHandle (h: Handle);
```

Trap macro	_EmptyHandle			
On entry	AØ:	h (handle)		
On exit	AØ:	h (handle)		
	DØ:	result code (integer)		
Result codes	Ø	$ØØØØ	noErr	No error
	-111	$FF91	memWZErr	Attempt to operate on a free block
	-112	$FF9Ø	memPurErr	Block is locked

EmptyHandle empties handle h: that is, it purges the relocatable block whose handle is h from its heap zone and sets its master pointer to NIL. If h is already empty, EmptyHandle does nothing.

(hand)

The main use of this procedure is to release the space a block occupies without having to update every existing handle to the block. Since the space occupied by the master pointer itself remains allocated, all handles pointing to it remain valid but become empty. When you later reallocate space for the block with ReallocHandle, the master pointer will be updated, causing all existing handles to point correctly to the new block.

The block whose handle is h must be unlocked, but need not be purgeable: if you ask to purge an unpurgeable block, EmptyHandle assumes you know what you're doing and purges the block as requested. If the block is locked, EmptyHandle doesn't purge it; after the call, MemError will return memPurErr. If the block is already free, MemError will return memWZErr.

Properties of Relocatable Blocks

```
PROCEDURE HLock (h: Handle);
```

Trap macro	_HLock			
On entry	A0:	h (handle)		
On exit	D0:	result code (integer)		
Result codes	0	$0000	noErr	No error
	-109	$FF93	nilHandleErr	NIL master pointer
	-111	$FF91	memWZErr	Attempt to operate on a free block

HLock locks a relocatable block, preventing it from being moved within its heap zone. After the call, MemError will return nilHandleErr if handle h is empty or memWZErr if it points to a free block, otherwise noErr. If the block is already locked, HLock does nothing.

```
PROCEDURE HUnlock (h: Handle);
```

Trap macro	_HUnlock			
On entry	A0:	h (handle)		
On exit	D0:	result code (integer)		
Result codes	0	$0000	noErr	No error
	-109	$FF93	nilHandleErr	NIL master pointer
	-111	$FF91	memWZErr	Attempt to operate on a free block

HUnlock unlocks a relocatable block, allowing it to be moved within its heap zone. After the call, MemError will return nilHandleErr if handle h is empty or memWZErr if it points to a free block, otherwise noErr. If the block is already unlocked, HUnlock does nothing.

```
PROCEDURE HPurge (h: Handle);
```

Trap macro	_HPurge			
On entry	A0:	h (handle)		
On exit	D0:	result code (integer)		
Result codes	0	$0000	noErr	No error
	-109	$FF93	nilHandleErr	NIL master pointer
	-111	$FF91	memWZErr	Attempt to operate on a free block

HPurge marks a relocatable block as purgeable. After the call, MemError will return nilHandleErr if handle h is empty or memWZErr if it points to a free block, otherwise noErr. If the block is already purgeable, HPurge does nothing.

```
PROCEDURE HNoPurge (h: Handle);
```

Trap macro	_HNoPurge			
On entry	A0:	h (handle)		
On exit	D0:	result code (integer)		
Result codes	0	$0000	noErr	No error
	-109	$FF93	nilHandleErr	NIL master pointer
	-111	$FF91	memWZErr	Attempt to operate on a free block

HNoPurge marks a relocatable block as unpurgeable. After the call, MemError will return nilHandleErr if handle h is empty or memWZErr if it points to a free block, otherwise noErr. If the block is already unpurgeable, HNoPurge does nothing.

Grow Zone Functions

PROCEDURE SetGrowZone (growZone: ProcPtr);

Trap macro	_SetGrowZone	
On entry	A0:	growZone (pointer)
On exit	D0:	result code (integer)
Result codes	0 $0000	noErr No error

SetGrowZone sets the current heap zone's grow zone function as designated by the growZone parameter. A NIL parameter value removes any grow zone function the zone may previously have had. After the call, MemError will always return noErr.

(hand)
If your program presses the limits of the available heap space, it's a good idea to have a grow zone function of some sort. At the very least, the grow zone function should detect when the Memory Manager is about to run out of space at a critical time (see GZCritical, below) and take some graceful action--such as displaying an alert box with the message "Out of memory"--instead of just failing unpredictably. *** There may eventually be a default grow zone function that does this. ***

The Memory Manager calls the grow zone function as a last resort when trying to allocate space, after failing to create a block of the needed size by compacting the zone, increasing its size (in the case of the original application zone), or purging blocks from it. Memory Manager routines that may cause the grow zone function to be called are NewHandle, NewPtr, SetHandleSize, SetPtrSize, ReallocHandle, and ResrvMem.

The grow zone function should be of the form

```
FUNCTION GrowTheZone (cbNeeded: Size) : Size;
```

(Of course, the name GrowTheZone is only an example; you can give the function any name you like.) The cbNeeded parameter gives the physical size of the needed block in bytes, **including the block header.** The grow zone function should attempt to create a free block of at least this size. It should return as its result the number of additional bytes it has freed within the zone, but this number need not be accurate.

If the grow zone function returns Ø, the Memory Manager will give up trying to allocate the needed block and will signal failure with the result code memFullErr. Otherwise it will compact the heap zone and try again to allocate the block. If still unsuccessful, it will continue to call the grow zone function repeatedly, compacting the zone again after each call, until it either succeeds in allocating the needed block or receives a zero result and gives up.

The usual way for the grow zone function to free more space is to call EmptyHandle to purge blocks that were previously marked unpurgeable. Another possibility is to unlock blocks that were previously locked, in order to eliminate immovable "islands" that may have been interfering with the compaction process and fragmenting the existing free space.

(hand)

Although just unlocking blocks doesn't actually free any additional space in the zone, the grow zone function should still return a nonzero result in this case. This signals the Memory Manager to compact the heap and try again to allocate the needed block.

(eye)

Depending on the circumstances in which the grow zone function is called, there may be particular blocks within the heap zone that must not be purged or released. For instance, if your program is attempting to increase the size of a relocatable block with SetHandleSize, it would be disastrous to release the block being expanded. To deal with such cases safely, it's essential to understand the use of the functions GZCritical and GZSaveHnd (see below).

FUNCTION GZCritical : BOOLEAN; [Pascal only]

Trap macro None

Result codes None

GZCritical returns TRUE if the Memory Manager critically needs the requested space: for example, to create a new relocatable or nonrelocatable block or to reallocate a handle. It returns FALSE in less critical cases, such as ResrvMem trying to move a block in order to reserve space as low as possible in the heap zone or SetHandleSize trying to increase the size of a relocatable block by moving the block above it.

(eye)

If you're writing a grow zone function in Pascal, you should always call GZCritical and proceed only if the result is TRUE. All the information you need to handle

the critical cases safely is the value of GZSaveHnd (see below). The noncritical cases require additional information that isn't available from Pascal, so your grow zone function should just return Ø and not attempt to free any space.

Assembly-language note: GZCritical is part of the Pascal interface to the Memory Manager, not part of the Memory Manager itself. It doesn't reside in ROM and can't be called via a trap. To find out whether a given grow zone call is critical, use the following magical incantation:

```
        MOVE.L  gzMoveHnd,DØ
        BEQ.S   Critical
        CMP.L   gzRootHnd,DØ
        BEQ.S   Critical

        CLR.L   4(SP)           ;If noncritical, just return Ø
        RTS

Critical . . .                  ;Handle critical case
```

To handle the critical cases safely (and the noncritical ones if you choose to do more than just return Ø), see the note below under GZSaveHnd.

FUNCTION GZSaveHnd : Handle; [Pascal only]

Trap macro None

Result codes None

GZSaveHnd returns a handle to a relocatable block that mustn't be purged or released by the grow zone function, or NIL if there is no such block. The grow zone function will be safe if it avoids purging or releasing this block, **provided that the grow zone call was critical.** To handle noncritical cases safely, further information is needed that isn't available from Pascal.

Assembly-language note: GZSaveHnd is part of the Pascal interface to the Memory Manager, not part of the Memory Manager itself. It doesn't reside in ROM and can't be called via a trap. You can find the handle it returns in the global variable gzRootHnd. The "further information" that isn't available from

Pascal is the contents of two other global variables, gzRootPtr and gzMoveHnd, which may be nonzero in noncritical cases. If gzRootPtr is nonzero, it's a pointer to a nonrelocatable block that must not be released; gzMoveHnd is a handle to a relocatable block that must not be released but may be purged.

Utility Routines

```
PROCEDURE BlockMove (sourcePtr,destPtr: Ptr; byteCount: Size);
```

Trap macro	_BlockMove		
On entry	A0:	sourcePtr (pointer)	
	A1:	destPtr (pointer)	
	D0:	byteCount (long integer)	
On exit	D0:	result code (integer)	
Result codes	0	$0000 noErr	No error

BlockMove moves a block of byteCount consecutive bytes from the address designated by sourcePtr to that designated by destPtr. No checking of any kind is done on the addresses; no pointers are updated. After the call, MemError will always return noErr.

```
FUNCTION TopMem : Ptr;  [Pascal only]
```

Trap macro	None		
Result codes	0	$0000 noErr	No error

TopMem returns a pointer to the address following the last byte of physical memory. After the call, MemError will always return noErr.

Assembly-language note: TopMem is part of the Pascal interface to the Memory Manager, not part of the Memory Manager itself. It doesn't reside in ROM and can't be called via a trap. To get a pointer to the end of physical memory from assembly language, use the global variable memTop.

FUNCTION MemError : MemErr; [Pascal only]

Trap macro None

Result codes None

MemError returns the result code produced by the last Memory Manager routine to be called.

Assembly-language note: MemError is part of the Pascal interface to the Memory Manager, not part of the Memory Manager itself. It doesn't reside in ROM and can't be called via a trap. To get the a routine's result code from assembly language, look in register DØ on return from the routine.

SPECIAL TECHNIQUES

This section describes some special or unusual techniques that you may find useful.

Dereferencing a Handle

Accessing a block by double indirection, through a handle instead of a simple pointer, requires an extra memory reference. For efficiency, you may sometimes want to dereference the handle--that is, convert it to a copy of the master pointer, then use that pointer to access the block by single indirection. But **be careful!** Any operation that allocates space from the heap may cause the underlying block to be moved or purged. In that event, the master pointer itself will be correctly updated, but your copy of it will be left dangling.

One way to avoid this common type of program bug is to lock the block before dereferencing its handle: for example,

```
VAR aPointer: Ptr;
    aHandle: Handle;
    . . . ;
BEGIN
    . . . ;
    aHandle := NewHandle( . . . );    {create a relocatable block}
    . . . ;
    HLock(aHandle);                   {lock block before dereferencing}
    aPointer := aHandle^;             {convert handle to simple pointer}

    WHILE . . . DO
        BEGIN
            ...aPointer^...           {use simple pointer inside loop}
        END;

    HUnlock(aHandle);                 {unlock block when finished}
    . . .
END
```

Assembly-language note: To dereference a handle in assembly language, just copy the master pointer into an address register and use it to access the block by single indirection. Remember that the master pointer points to the block's **contents**, not its header!

```
          MOVE.L  #blockSize,DØ ;set up block size for _NewHandle
          _NewHandle            ;create relocatable block
          MOVE.L  AØ,aHandle    ;save handle for later use
          . . .
          MOVE.L  aHandle,A1    ;get back handle
          MOVE.L  A1,AØ         ;lock block before dereferencing
          _HLock

          MOVE.L  (A1),A2       ;convert handle to simple pointer

LOOP      . . .
          MOVE    ...(A2)...    ;use simple pointer inside loop
          . . .
          Bcc.S   LOOP          ;loop back on some condition

          MOVE.L  A1,AØ         ;unlock block when finished
          _HUnlock
          . . .
```

Remember, however, that when you lock a block it becomes an "island" in the heap that may interfere with compaction and cause free space to become fragmented. It's recommended that you use this technique only in parts of your program where efficiency is critical, such as inside tight inner loops that are executed many times.

(eye)

Don't forget to unlock the block again when you're through with the dereferenced handle!

Instead of locking the block, you can update your copy of the master pointer after any "dangerous" operation (one that can invalidate the pointer by moving or purging the block it points to). Memory Manager routines that can move or purge blocks in the heap are NewHandle, NewPtr, SetHandleSize, SetPtrSize, ReallocHandle, ResrvMem, CompactMem, PurgeMem, and MaxMem. Since these routines can be called indirectly from other Operating System or Toolbox routines, you should assume that any call to the OS or Toolbox can potentially leave your dereferenced pointer dangling. *** Eventually there will be a technical note listing which OS and Toolbox routines are dangerous and which aren't. ***

(hand)

If you aren't performing any dangerous operations, you needn't worry about updating the pointer (or locking the block either, for that matter).

Subdividing the Application Heap Zone

In some applications, you may want to subdivide the original application heap zone into two or more independent zones to be used for different purposes. In doing this, it's important not to destroy any existing blocks in the original zone (such as those containing the code of your program). The recommended procedure is to allocate space for the subzones as nonrelocatable blocks within the original zone, then use InitZone to initialize them as independent zones. For example, to divide the available space in the application zone in half, you might write something like the following:

```
CONST minSize = 52 + 12 + 32*(12 + 4);    {zone header, zone trailer,}
                                          {   and 32 minimum-size blocks}
                                          {   with master pointers}
VAR myZone1, myZone2: THz;
    start, limit: Ptr;
    availSpace, zoneSize: Size;
    . . . ;
BEGIN
    . . . ;
    SetZone(ApplicZone);
    availSpace := CompactMem(maxSize);    {size of largest free block}
    zoneSize := 2 * (availSpace DIV 4);   {force new zone size to an}
                                          {   even number of bytes}
    IF zoneSize < (minSize + 8)           {need 8 bytes for}
                                          {   block header}
       THEN . . .                         {error--not enough room}
       ELSE
          BEGIN
             zoneSize := zoneSize - 8;    {adjust for block header}

             start := NewPtr(zoneSize);   {allocate a nonrel. block}
             limit := POINTER(ORD(start) + zoneSize);
             InitZone(NIL, 32, limit, start);
             myZone1 := POINTER(ORD(start));  {convert Ptr to THz}

             start := NewPtr(zoneSize);   {allocate a nonrel. block}
             limit := POINTER(ORD(start) + zoneSize);
             InitZone(NIL, 32, limit, start);
             myZone2 := POINTER(ORD(start))   {convert Ptr to THz}
          END;
    . . .
END
```

Assembly-language note: The equivalent assembly code might be

```
minSize   .EQU    52+12+<32*<12+4>> ;zone header and trailer, plus
                                    ;   32 minimum-size blocks
                                    ;   with master pointers
          . . .
          MOVE.L  applZone,AØ       ;get original application zone
          _SetZone                  ;make it current

          MOVE.L  #maxSize,DØ       ;compact entire zone
          _CompactMem               ;DØ has size of largest free block

          ASR.L   #2,DØ             ;force new zone size to an
          ASL.L   #1,DØ             ;   even number of bytes
          CMP.L   #minSize+8,DØ     ;need 8 bytes for block header
          BLO     NoRoom            ;error if < minimum size

          SUBQ.L  #8,DØ             ;adjust for block header
          MOVE.L  DØ,D1             ;save zone size
          _NewPtr                   ;allocate nonrelocatable block
          MOVE.L  AØ,myZone1        ;store zone pointer

          CLR.L   -(SP)             ;NIL grow zone function
          MOVE.W  #32,-(SP)         ;allocate 32 master pointers
          MOVE.L  AØ,-(SP)          ;AØ has zone pointer
          ADD.L   D1,(SP)           ;convert to limit pointer
          MOVE.L  AØ,-(SP)          ;push as start pointer

          MOVE.L  SP,AØ             ;point to argument block
          _InitZone                 ;create zone 1

          MOVE.L  D1,DØ             ;get back zone size
          _NewPtr                   ;allocate nonrelocatable block
          MOVE.L  AØ,myZone2        ;store zone pointer

          MOVE.L  AØ,4(SP)          ;move zone pointer to stack
          ADD.L   D1,(SP)           ;convert to limit pointer
          MOVE.L  AØ,(SP)           ;move to stack as start pointer

          MOVE.L  SP,AØ             ;point to argument block
          _InitZone                 ;create zone 2
          ADD.W   #14,SP            ;pop arguments off stack
          . . .
```

Creating a Heap Zone on the Stack

Another place you can get the space for a new heap zone is from the stack. For example,

```
CONST zoneSize = 2Ø48;
VAR zoneArea: PACKED ARRAY [1..zoneSize] OF SignedByte;
    stackZone: THz;
    limit: Ptr;
    . . . ;
BEGIN
    . . . ;
    stackZone := @zoneArea;
    limit := POINTER(ORD(stackZone) + zoneSize);
    InitZone(NIL, 16, limit, @zoneArea);
    . . .
END
```

Assembly-language note: Here's how you might do the same thing in assembly language:

```
zoneSize  .EQU    2Ø48
          . . .
          MOVE.L  SP,A2           ;save stack pointer for limit
          SUB.W   #zoneSize,SP    ;make room on stack
          MOVE.L  SP,A1           ;save stack pointer for start
          MOVE.L  A1,stackZone    ;store as zone pointer

          CLR.L   -(SP)           ;NIL grow zone function
          MOVE.W  #16,-(SP)       ;allocate 16 master pointers
          MOVE.L  A2,-(SP)        ;push limit pointer
          MOVE.L  A1,-(SP)        ;push start pointer

          MOVE.L  SP,AØ           ;point to argument block
          _InitZone               ;create new zone
          ADD.W   #14,SP          ;pop arguments off stack
          . . .
```

NOTES FOR ASSEMBLY-LANGUAGE PROGRAMMERS

General information about how to use the Macintosh Operating System from assembly language is *** (will be) *** given elsewhere. This section contains special notes of interest to programmers who will be using the Memory Manager from assembly language.

The primary aids to assembly-language programmers are files named SYSEQU.TEXT, SYSMACS.TEXT, SYSERR.TEXT, and HEAPDEFS.TEXT. If you use .INCLUDE to include these files when you assemble your program, all the Memory Manager constants, addresses of global variables, trap macros, error codes, and masks and offsets into fields of structured types will be available in symbolic form.

Constants

The file HEAPDEFS.TEXT defines a number of useful constants that you can use in your program as immediate data values. For example, to push the default master-point count onto the stack as an argument for _InitZone, you might write

```
MOVE.W  #dfltMasters,-(SP)
```

(hand)

It's a good idea to refer to these constants in your program by name instead of using the numeric value directly, since some of the values shown may be subject to change. Some of the constants are based on an eventual 512K memory configuration; the present Macintosh has 128K of RAM.

The following constants are defined in HEAPDEFS.TEXT:

```
minFree        .EQU    12          ;minimum block size
maxSize        .EQU    $7FFFF      ;maximum block size (512K - 1)
minAddr        .EQU    Ø           ;minimum legal address
maxAddr        .EQU    $8ØØØØ      ;maximum legal address (512K)
dfltMasters    .EQU    32          ;default master-pointer count
maxMasters     .EQU    $1ØØØ       ;maximum master-pointer count (4K)
sysZoneSize    .EQU    $4ØØØ       ;size of system heap zone (16K)
applZoneSize   .EQU    $18ØØ       ;initial size of application zone (6K)
minZone        .EQU    heapData+<4*minFree>+<8*dfltMasters>
                                   ;minimum size of application zone
dfltStackSize .EQU     $ØØØØ2ØØØ ;initial space allotment for stack (8K)

tybkFree       .EQU    Ø           ;tag value for free block
tybkNRel       .EQU    1           ;tag value for nonrelocatable block
tybkRel        .EQU    2           ;tag value for relocatable block
```

One global constant pertinent to the Memory Manager is defined in SYSEQU.TEXT:

```
heapStart     .EQU    $ØBØØ     ;start address of
                                ;   system heap zone (2816)
```

Global Variables

The Memory Manager's global variables are located in the system communication area and defined in the file SYSEQU.TEXT. To access a global variable, just refer to it by name as an absolute address. For example, to load a pointer to the current heap zone into register A2, write

```
MOVE.L  theZone,A2
```

The following global variables are used by the Memory Manager:

Variable	Contents
memTop	Limit address (end plus one) of physical memory
bufPtr	Base address of stack (grows downward from here)
minStack	Minimum space allotment for stack (1K)
defltStack	Default space allotment for stack (8K)
heapEnd	Current limit address of application heap zone
applLimit	Application heap limit
sysZone	Address of system heap zone
applZone	Address of application heap zone
theZone	Address of current heap zone

Trap Macros

All assembly-language trap macros for the Memory Manager (as well as the rest of the Operating System) are defined in the file SYSMACS.TEXT. To call a Memory Manager routine from assembly language via the trap mechanism, just use the name of the trap macro as the operation code of an instruction. For example, to find out the number of free bytes in the current heap zone, use the instruction

```
_FreeMem
```

As stated in the description of FreeMem above, the number of free bytes will be in register DØ on return from the trap.

Result Codes

The file SYSERR.TEXT contains constant definitions for all result codes returned by Operating System routines. You can use them in your program as immediate data values. For example, to test for the error code memFullErr on return from a trap, you might write

```
CMP.W   #memFullErr,DØ
BEQ     NoRoom
```

The Memory Manager uses the following error codes:

```
noErr          .EQU    Ø          ;no error
memFullErr     .EQU    -1Ø8       ;not enough room in zone
nilHandleErr   .EQU    -1Ø9       ;NIL master pointer
memWZErr       .EQU    -111       ;attempt to operate on a free block
memPurErr      .EQU    -112       ;attempt to purge a locked block
```

Offsets and Masks

Offsets to the fields of zone and block headers are defined as constants in the file HEAPDEFS.TEXT. To access a field, use the name of the offset constant as a displacement relative to an address register pointing to the first byte of the header. For example, if register A2 contains a pointer to a zone header, you can load the number of free bytes in the zone into D3 with the instruction

```
MOVE.L  gzProc(A2),D3
```

(eye)

> Generally speaking, the offset and mask constants discussed here are intended for the Memory Manager's internal use. You shouldn't ordinarily be prowling around in a zone or block header unless you know what you're doing.

The following offset constants represent the fields of a zone header:

```
bkLim          .EQU    Ø          ;address of zone trailer (long)
purgePtr       .EQU    4          ;roving purge pointer (long)
hFstFree       .EQU    8          ;address of first free
                                  ;   master pointer (long)
zcbFree        .EQU    12         ;number of free bytes (long)
gzProc         .EQU    16         ;address of grow zone
                                  ;   function (long)
moreMasters    .EQU    2Ø         ;incremental master-pointer
                                  ;   count (word)
flags          .EQU    22         ;internal flags (word)
cntRel         .EQU    24         ;relocatable blocks (word)
```

```
maxRel      .EQU    26      ;max. cntRel so far (word)
cntNRel     .EQU    28      ;nonrelocatable blocks (word)
maxNRel     .EQU    30      ;max. cntNRel so far (word)
cntEmpty    .EQU    32      ;empty master pointers (word)
cntHandles  .EQU    34      ;total master pointers (word)
minCBFree   .EQU    36      ;min. zcbFree so far (long)
purgeProc   .EQU    40      ;address of purge warning
                            ;  procedure (long)
sparePtr    .EQU    44      ;spare pointer (long)
allocPtr    .EQU    48      ;roving allocation pointer (long)
heapData    .EQU    52      ;first usable byte in zone
```

The following offset constants represent the fields of a block header:

```
tagBC       .EQU    0       ;tag, size correction, and
                            ;   physical byte count (long)
handle      .EQU    4       ;reloc.:  relative handle (long)
                            ;nonreloc.:  zone pointer (long)
blkData     .EQU    8       ;first byte of block contents
```

HEAPDEFS.TEXT also defines the following mask constants for manipulating the fields of block headers and master pointers:

```
tagMask     .EQU    $C0000000 ;tag field
bcOffMask   .EQU    $0F000000 ;size correction
                              ;   ("byte count offset")
bcMask      .EQU    $00FFFFFF ;physical byte count
ptrMask     .EQU    $00FFFFFF ;address part of master pointer
                              ;   or zone pointer
handleMask  .EQU    $00FFFFFF ;relative handle
freeTag     .EQU    0         ;tag for free block
nRelTag     .EQU    $40000000 ;tag for nonrelocatable block
relTag      .EQU    $80000000 ;tag for relocatable block
```

(eye)

Remember, the pointer or handle you get from the Memory Manager when you allocate a block points to the block's **contents**, not its header. To get the address of the header, subtract the offset constant blkData, defined above. For example, if you have a handle to a block in register A2, the following code will set A3 to point to the block's header:

```
MOVE.L  (A2),A3         ;get pointer to block contents
SUBQ.L  #blkData,A3     ;offset back to header
```

Finally, SYSEQU.TEXT defines the following constants for the bit numbers of the various flag bits within the high-order byte of a master pointer:

```
lock        .EQU    7       ;lock bit
purge       .EQU    6       ;purge bit
resource    .EQU    5       ;resource bit
```

You can use these constants to access the flag bits directly, using the 68ØØØ instructions BSET, BCLR, and BTST. For instance, if you have a handle to a relocatable block in register A2, you can mark the block as purgeable with the instruction

```
BSET.B  #purge,(A2)   ;set purge bit in master pointer
```

To branch on the current setting of the lock bit,

```
BTST.B  #lock,(A2)    ;test lock bit in master pointer
BNE     ItsLocked     ;   and branch on result
```

Handy Tricks

To save time in critical situations, here's a quick way to convert a dereferenced pointer to a relocatable block back into a handle without paying the overhead of a _RecoverHandle trap. Recall that the relative handle stored in the block's header is the offset of the block's master pointer relative to the start of its heap zone. So to convert a copy of the master pointer back into the original handle, find the relative handle and add it to the address of the zone. For example, if register A2 contains the master pointer of a block in the current heap zone, the following code will reconstruct the block's handle in A3:

```
MOVE.L  -4(A2),A3     ;relative handle is 4 bytes back
                      ;   from start of contents
ADD.L   theZone,A3    ;use as offset from start of zone
```

Conversely, given a true (absolute) handle to a relocatable block, you can find the zone the block belongs to by subtracting the relative handle from the absolute handle. If the absolute handle is in register A2, the following instructions will convert it into a pointer to the block's heap zone:

```
MOVE.L  (A2),A3       ;get pointer to block
SUB.L   -4(A3),A2     ;subtract relative handle
                      ;   to get zone pointer
```

For nonrelocatable blocks, the header contains a pointer directly back to the zone:

```
MOVE.L  -4(A2),A2     ;get zone pointer directly
```

SUMMARY OF THE MEMORY MANAGER

```
CONST noErr         = Ø;       {no error}
      memFullErr    = -1Ø8;    {not enough room in zone}
      nilHandleErr  = -1Ø9;    {NIL master pointer}
      memWZErr      = -111;    {attempt to operate on a free block}
      memPurErr     = -112;    {attempt to purge a locked block}

      maxSize = $8ØØØØØ;

TYPE SignedByte = -128..127;
     Byte       = Ø..255;
     Ptr        = ^SignedByte;
     Handle     = ^Ptr;
     ProcPtr    = Ptr;

     Size   = LongInt;
     MemErr = INTEGER;

     THz  = ^Zone;
     Zone = RECORD
              bkLim:      Ptr;
              purgePtr:   Ptr;
              hFstFree:   Ptr;
              zcbFree:    LongInt;
              gzProc:     ProcPtr;
              moreMast:   INTEGER;
              flags:      INTEGER;
              cntRel:     INTEGER;
              maxRel:     INTEGER;
              cntNRel:    INTEGER;
              maxNRel:    INTEGER;
              cntEmpty:   INTEGER;
              cntHandles: INTEGER;
              minCBFree:  LongInt;
              purgeProc:  ProcPtr;
              sparePtr:   Ptr;
              allocPtr:   Ptr;
              heapData:   INTEGER
            END;
```

Initialization and Allocation

```
PROCEDURE InitApplZone;
PROCEDURE SetApplBase   (startPtr: Ptr);
PROCEDURE InitZone      (growProc: ProcPtr; masterCount: INTEGER;
                         limitPtr, startPtr: Ptr);
PROCEDURE SetApplLimit  (zoneLimit: Ptr);
```

Heap Zone Access

```
FUNCTION  GetZone :    THz;
PROCEDURE SetZone     (hz: THz);
FUNCTION  SystemZone : THz;  [Pascal only]
FUNCTION  ApplicZone : THz;  [Pascal only]
```

Allocating and Releasing Relocatable Blocks

```
FUNCTION  NewHandle     (logicalSize: Size) : Handle;
PROCEDURE DisposHandle  (h: Handle);
FUNCTION  GetHandleSize (h: Handle) : Size;
PROCEDURE SetHandleSize (h: Handle; newSize: Size);
FUNCTION  HandleZone    (h: Handle) : THz;
FUNCTION  RecoverHandle (p: Ptr) : Handle;
PROCEDURE ReallocHandle (h: Handle; logicalSize: Size);
```

Allocating and Releasing Nonrelocatable Blocks

```
FUNCTION  NewPtr     (logicalSize: Size) : Ptr;
PROCEDURE DisposPtr  (p: Ptr);
FUNCTION  GetPtrSize (p: Ptr) : Size;
PROCEDURE SetPtrSize (p: Ptr; newSize: Size);
FUNCTION  PtrZone    (p: Ptr) : THz;
```

Freeing Space on the Heap

```
FUNCTION  FreeMem :    LongInt;
FUNCTION  MaxMem      (VAR grow: Size) : Size;
FUNCTION  CompactMem  (cbNeeded: Size) : Size;
PROCEDURE ResrvMem    (cbNeeded: Size);
FUNCTION  PurgeMem    (cbNeeded: Size);
PROCEDURE EmptyHandle (h: Handle);
```

Properties of Relocatable Blocks

```
PROCEDURE HLock    (h: Handle);
PROCEDURE HUnlock  (h: Handle);
PROCEDURE HPurge   (h: Handle);
PROCEDURE HNoPurge (h: Handle);
```

Grow Zone Functions

```
PROCEDURE SetGrowZone (growZone: ProcPtr);
FUNCTION  GZCritical : BOOLEAN;  [Pascal only]
FUNCTION  GZSaveHnd :  Handle;   [Pascal only]
```

Utility Routines

```
PROCEDURE BlockMove (sourcePtr, destPtr: Ptr; byteCount: Size);
FUNCTION  TopMem :   Ptr;      [Pascal only]
FUNCTION  MemError : MemErr;   [Pascal only]
```

GLOSSARY

allocate: To reserve a block for use.

application heap zone: The heap zone provided by the Memory Manager for use by the application program.

block: An area of contiguous memory within a heap zone.

block contents: The area of a block available for use.

block header: The internal "housekeeping" information maintained by the Memory Manager at the beginning of each block in a heap zone.

compaction: The process of moving allocated blocks within a heap zone in order to collect the free space into a single block.

current heap zone: The heap zone currently under attention, to which most Memory Manager operations implicitly apply.

dereference: To convert a pointer into whatever it points to; specifically, to convert a handle into a copy of its corresponding master pointer.

empty handle: A handle that points to a NIL master pointer, signifying that the underlying relocatable block has been purged.

free block: A block containing space available for allocation.

grow zone function: A function supplied by the application program to help the Memory Manager create free space within a heap zone.

handle: A pointer to a master pointer, which designates a relocatable block by double indirection.

heap zone: An area of memory in which space can be allocated and released on demand, using the Memory Manager.

limit pointer: A pointer to the byte following the last byte of an area in memory, such as a block or a heap zone.

lock: To temporarily prevent a relocatable block from being moved during heap compaction.

lock bit: A bit in the master pointer to a relocatable block that indicates whether the block is currently locked.

logical size: The number of bytes in a block's contents; compare physical size.

master pointer: A single pointer to a relocatable block, maintained by the Memory Manager and updated whenever the block is moved, purged, or reallocated. All handles to a relocatable block refer to it by double indirection through the master pointer.

nonrelocatable block: A block whose location in its heap zone is fixed and can't be moved during heap compaction.

physical size: The actual number of bytes a block occupies within its heap zone.

purge: To remove a relocatable block from its heap zone, leaving its master pointer allocated but set to NIL.

purgeable block: A relocatable block that can be purged from its heap zone.

purge bit: A bit in the master pointer to a relocatable block that indicates whether the block is currently purgeable.

purge warning procedure: A procedure associated with a particular heap zone that is called whenever a block is purged from that zone.

reallocate: To allocate new space in a heap zone for a purged block, updating its master pointer to point to its new location.

relative handle: A handle to a relocatable block expressed as the offset of its master pointer within the heap zone, rather than as the absolute memory address of the master pointer.

release: To destroy an allocated block, freeing the space it occupies.

relocatable block: A block that can be moved within its heap zone during compaction.

result code: An integer code produced by a Memory Manager routine to signal the success of an operation or the reason for its failure.

size correction: The number of unused bytes included at the end of an allocated block; the difference between the block's logical and physical sizes, excluding the block header.

system heap zone: The heap zone provided by the Memory Manager for use by the Macintosh system software.

tag: A 2-bit code in the header of a block identifying it as relocatable, nonrelocatable, or free.

unlock: To allow a relocatable block to be moved during heap compaction.

unpurgeable block: A relocatable block that can't be purged from its heap zone.

zone header: The internal "housekeeping" information maintained by the Memory Manager at the beginning of each heap zone.

zone pointer: A pointer to a zone record.

zone record: A Pascal data structure representing the structure of a zone header.

zone trailer: A minimum-size free block marking the end of a heap zone.

COMMENTS?

Macintosh User Education encourages your comments on this manual.

- What do you like or dislike about it?

- Were you able to find the information you needed?

- Was it complete and accurate?

- Do you have any suggestions for improvement?

Please send your comments to the author (indicated on the cover page) at 10460 Bandley Drive M/S 3-G, Cupertino CA 95014. Mark up a copy of the manual or note your remarks separately. (We'll return your marked-up copy if you like.)

Thanks for your help!

MACINTOSH USER EDUCATION

The Segment Loader: A Programmer's Guide /SEGLOAD/SEGMENT

See Also: Macintosh Operating System Reference Manual
The Resource Manager: A Programmer's Guide
The Macintosh Finder

Modification History: First Draft (ROM 4) C. Rose 6/24/83

ABSTRACT

This manual describes the Segment Loader of the Macintosh Operating System, which lets you divide your application into several parts and have only some of them in memory at a time.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Segment Loader, a new part of the Macintosh Operating System in ROM version 4. *** Eventually it will become part of a large manual describing the entire Operating System and Toolbox. *** The Segment Loader lets you divide your application into several parts and have only some of them in memory at a time.

You should already be familiar with Lisa Pascal, the Macintosh Operating System's Memory Manager, the Finder, and the basic concepts behind the Resource Manager of the Macintosh User Interface Toolbox.

The manual begins with an introduction to the Segment Loader and a description of the parameters that are stored in memory when an application is started up. Next, a section on using the Segment Loader introduces you to its routines and tells how they fit into the flow of your application. This is followed by the detailed descriptions of all Segment Loader routines, their parameters, calling protocol, effects, side effects, and so on.

For advanced programmers, there's a section that discusses the jump table, explaining how the Segment Loader works internally.

Finally, there's a summary of the Segment Loader routine calls, for quick reference, and a glossary of terms defined in this manual.

ABOUT THE SEGMENT LOADER

The Segment Loader allows you to divide the code of an application into several parts or segments. The Finder starts up an application by calling a Segment Loader routine that loads in the main segment (the one containing the main program). Other segments are loaded in automatically when they're needed. Your application can call the Segment Loader to have these other segments removed from memory when they're no longer needed.

The Segment Loader enables you to have programs larger than 32K bytes, the maximum size of a single segment. Also, any code that isn't executed often (such as code for printing hardcopy) need not occupy memory when it isn't being used, but can instead be in a separate segment that's brought in when needed.

This mechanism may remind you of the resources of an application, which the Resource Manager of the User Interface Toolbox reads into memory when necessary. An application's segments are in fact themselves stored as resources; their resource type is 'CODE'. You can use the Resource Compiler to create these resources from your application code. A "loaded" segment has been read into memory by the Resource Manager and locked (so that it's neither relocatable nor purgeable). When a segment is unloaded, it's made relocatable and purgeable.

Every segment has a name. If you do nothing about dividing your program into segments, it will consist of a single segment whose name is blank. Dividing your program into segments means specifying in your source file the beginning of each segment by name. The names are for your use only; they're not kept around after linking.

(eye)
If you do specify segment names, note that normally the main segment should have a blank name. The reason for this is that the intrinsic Pascal routines must be in the same segment as your main program, and the Linker puts those routines in the blank-named segment (so that the right thing will happen if you don't specify any segment names at all).

APPLICATION PARAMETERS

When an application is started up, certain parameters are stored in 32 bytes of memory just above the application's globals, as shown in Figure 1; these are called the application parameters. A5 points to the first of these parameters and may be used with positive offsets to access the others.

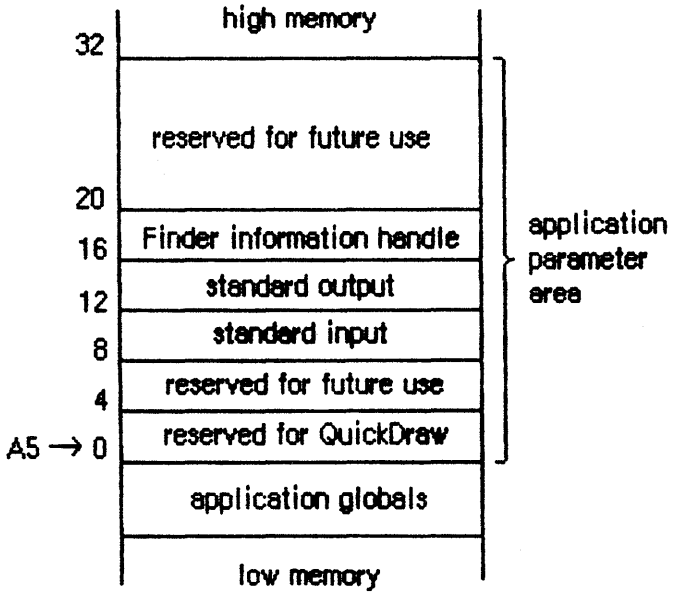


Figure 1. Application Parameters

(hand)
For brevity, we'll say "A5" where we mean "the location pointed to by A5".

The "standard input" and "standard output" parameters indicate the main source of input and destination of output for the Macintosh. They are

usually Ø, meaning the keyboard and the screen, respectively.

The "Finder information handle" is a handle to information that the Finder provides to the application upon starting it up. For example, for a word processor it might be the name of the document to be worked on. *** The exact information will be described here when available. *** Pascal programmers can call the Segment Loader routine GetAppParms to get the Finder information handle.

The other locations in the application parameter area are reserved for future use or for use by QuickDraw.

USING THE SEGMENT LOADER

This section introduces you to the Segment Loader routines and how they fit into the flow of an application program. The routines themselves are described in detail in the next section.

The routine that applications will most commonly use is UnloadSeg, for unloading a particular segment when it's no longer needed. Another useful routine, GetAppParms, lets you get information about your application such as its name and the reference number for its resources. For applications started up in the usual way by the Finder, GetAppParms also gives the Finder information handle that's stored 16 bytes above A5.

The main segment can unload other segments, but it can't get rid of itself; using the Chain routine, however, it can do something close to this. Chain starts up another application without disturbing the application heap. Thus the current application can let another application take over while still keeping its data around in the heap.

The Segment Loader also provides a quick exit to the Finder that doesn't touch the stack, for applications needing it in emergency situations: ExitToShell.

Finally, there are two advanced routines that most applications will never use: Launch and LoadSeg. Launch is called by the Finder to start up an application; it's like Chain but doesn't retain the application heap. LoadSeg is called indirectly (via the jump table, as described later) to load segments when necessary--that is, whenever a routine in an unloaded segment is invoked.

SEGMENT LOADER ROUTINES

This section describes all the Segment Loader routines. Some of the routines are stack-based and so are shown in Pascal; for information on using them from assembly language, see "Using the Toolbox from Assembly Language" *** doesn't exist, but see QuickDraw manual ***. Other Segment Loader routines are register-based and are described similar to

the way the Operating System routines are described in the current Operating System manual.

PROCEDURE UnloadSeg (routineAddr: Ptr);

UnloadSeg unloads a segment, making it relocatable and purgeable; routineAddr is the address of any routine in the segment. The Segment Loader will reload the segment the next time one of the routines in it is called. It doesn't hurt to call UnloadSeg, because the segment won't actually be purged until the memory it occupies is needed. If you need the unloaded segment again before it's purged, the Segment Loader won't have to access the disk.

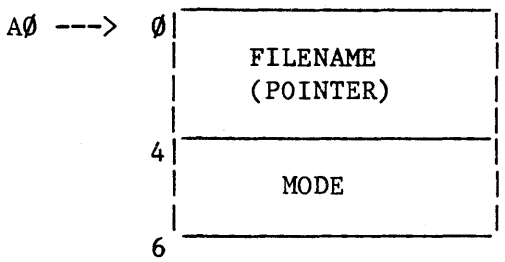
PROCEDURE GetAppParms (VAR apName: Str255; VAR apRefNum: INTEGER; VAR apParam: Handle);

GetAppParms returns information about the current application. It returns the application name in apName and the reference number for the application's resources in apRefNum. For applications started up in the usual way by the Finder, it returns the Finder information handle in apParam (as described earlier under "Application Parameters").

(hand)
For applications started up with the Chain routine (below), the apParam parameter isn't useful.

Chain {register-based}

This routine starts an application up without doing anything to the application heap, so the current application can let another application take over while still keeping its data around in the heap. It configures memory for the sound and video buffers. AØ points to the following:



where FILENAME is a pointer to the application's file name

and MODE tells which sound buffer and video buffer to use (Ø for standard).

The sound and video buffers are constantly scanned by the Macintosh hardware to determine what sounds to emit from its speakers and what to display on its screen. (The video buffer is the bit image corresponding to the display screen.) Two of each type of buffer are available; Figure 2 shows where they're located. If you specify a MODE

value of Ø, you get the standard or "primary" buffers; in this case, the application space begins where shown in Figure 2. Any positive MODE value causes the secondary sound buffer and primary video buffer to be used (which costs 1.5K of memory). Any negative MODE value causes the secondary sound buffer and secondary video buffer to be used (which costs 32K of memory).

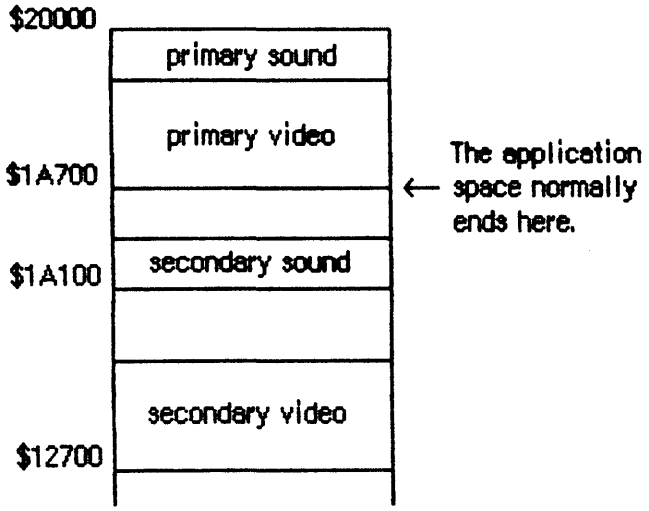


Figure 2. Sound and Video Buffers

Chain closes the resource file for any previous application and opens the resource file for the application being started. It also stores in memory the application parameters designating standard input and standard output. The application is started at its entry point, which causes the main segment to be loaded.

PROCEDURE ExitToShell;

ExitToShell provides an emergency exit for the application, without touching the stack. It simply launches the Finder (starts it up after freeing the storage occupied by the application heap; see Launch below).

Advanced Routines

Launch {register-based}

This routine is called by the Finder to start up an application and will rarely need to be called by an application itself. It's the same as the Chain routine (described above) except that it frees the storage occupied by the application heap and restores the heap to its original size. Also, the Finder provides startup information needed by the application; a handle to the information is located in the system heap

and is copied (as the "Finder information handle") into the application parameter area in memory.

(hand)
Launch preserves a special handle in the application heap which is used for accessing the scrap between applications.

PROCEDURE LoadSeg (segID: INTEGER);

LoadSeg is called indirectly via the jump table (as described in the following section) when the application calls a routine in an unloaded segment. It loads the segment having the given ID number, which was assigned by the Linker. If the segment isn't in memory, LoadSeg calls the Resource Manager to read it in. It changes the jump table entries for all the routines in the segment from the "unloaded" to the "loaded" state and then invokes the routine that was called.

THE JUMP TABLE

This section describes how the Segment Loader works internally, and is included here for advanced programmers; you don't have to know about this to be able to use the common Segment Loader routines.

The loading and unloading of segments is implemented through the application's jump table. Figure 3 shows the location of the jump table in memory for a typical application.

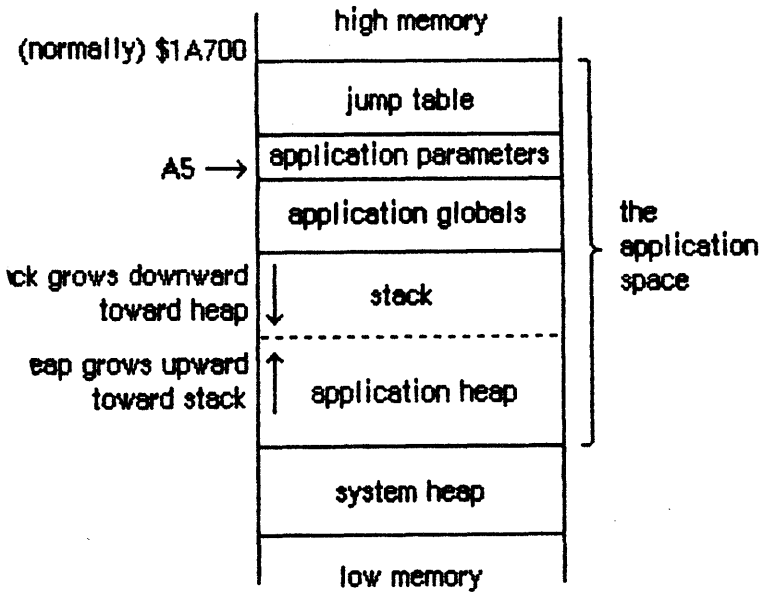


Figure 3. The Application's Space in Memory

When the Linker encounters a call to a routine in another segment, it creates a jump table entry for the routine and addresses the entry with a positive offset from A5. As described below, the jump table entry makes the connections necessary to invoke the routine.

The jump table contains one 8-byte entry for every externally referenced routine in every segment; all the entries for a particular segment are stored contiguously. It refers to segments by ID numbers assigned by the Linker. When an application is started up, its jump table is read in from segment Ø, a special segment created by the Linker for every executable file. Segment Ø contains the following:

Number of bytes	Contents
4 bytes	"Above A5" size; size in bytes from A5 to upper end of application space
4 bytes	"Below A5" size; size in bytes of application globals
4 bytes	Offset of jump table from A5
4 bytes	Length of jump table in bytes
n bytes	Jump table

For most applications, the offset of the jump table from A5 is 32, and the "above A5" size is 32 plus the length of the jump table.

All the jump table entries for a particular segment indicate whether that segment is currently loaded or not, as illustrated in Figure 4.

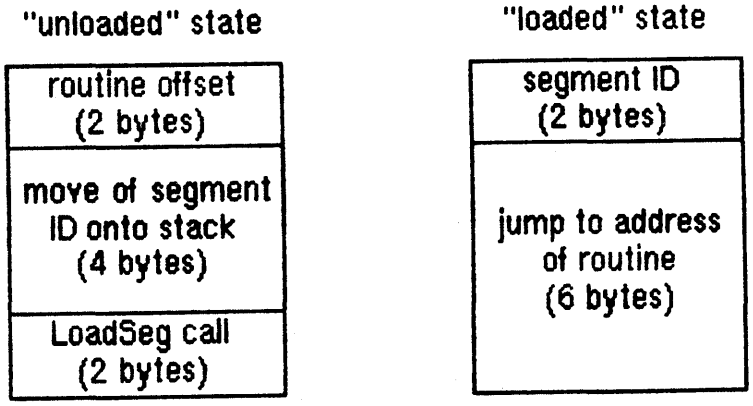


Figure 4. Format of a Jump Table Entry

Initially, of course, the jump table entries are all in the "unloaded" state, which means they contain the following:

Number of bytes	Contents
2 bytes	Offset of this routine from beginning of segment
4 bytes	Instruction that moves the segment ID onto the stack for LoadSeg
2 bytes	Trap that executes LoadSeg

When a call to a routine in an unloaded segment is made, the code in the last six bytes of its jump table entry is executed. This code calls LoadSeg, which loads the segment into memory, transforms all of its jump table entries to the "loaded" state (shown below), and invokes the routine.

Number of bytes	Contents
2 bytes	Segment ID
6 bytes	Instruction that jumps to the address of the routine for which this is an entry

LoadSeg invokes the routine by executing the instruction in the last six bytes of the jump table entry. Subsequent calls to the routine also execute this instruction. If UnloadSeg is called to unload the segment, it restores the jump table entries to their "unloaded" state. Notice that whether the segment is loaded or unloaded, the last six bytes of the jump table entry are executed; the effect depends on the state of the entry at the time.

To be able to set all the jump table entries for a segment to a particular state, LoadSeg and UnloadSeg need to know exactly where all the entries are located. They get this information from the segment header, four bytes at the beginning of the segment which contain the following:

Number of bytes	Contents
2 bytes	Offset of the first routine's entry from the beginning of the jump table
2 bytes	Number of entries for this segment

As described above, segment Ø tells where the beginning of the jump table is located.

SPECIFYING SEGMENTS IN YOUR SOURCE FILE

*** This section will be moved into the next version of the manual entitled "Putting Together a Macintosh Application". ***

You specify the beginning of a segment in your application's source file as follows:

```
{$S segname}
```

where segname is the segment name, a sequence of up to eight characters. Normally you should give the main segment a blank name.

For example, you might structure your program as follows:

```
PROGRAM Shell;

{ The USES statement and your LABEL, CONST, and VAR declarations
  will be here. }

{$S Seg1}

{ The procedures and functions in Seg1 will be here. }

{$S Seg2}

{ The procedures and functions in Seg2 will be here. }

{$S     }

BEGIN

  { The main program will be here. }

END.
```

You can specify the same segment name more than once; the routines will just be accumulated into that segment. To avoid problems when moving routines around in the source file, some programmers follow the practice of putting a segment name specification before every routine.

(eye)
Uppercase and lowercase letters ARE distinguished in segment names. For example, "Seg1" and "SEG1" are not equivalent names.

If you don't specify a segment name before the first routine in your file, the blank segment name will be assumed there.

In assembly language, you specify the beginning of a segment with the following directive:

```
.SEG 'segname'
```

(eye)
This requires version 12.2 of the Lisa Monitor.

You can also specify what segment the routines in a particular file should be in by using the ChangeSeg program. For example, suppose you want to give your main segment a nonblank name (say, "SegMain"); you can't do this without using ChangeSeg, because the Linker puts the intrinsic Pascal routines in the blank-named segment, and they must be in the same segment as your main program. You can use ChangeSeg as shown below to tell the Linker to put the intrinsic Pascal routines, which are in Obj:MacPasLib, in the segment named SegMain.

Prompt	Response
Monitor command line	X {for X(ecute}
What file ?	ChangeSeg <ret>
File to change:	Obj:MacPasLib <ret>
Map all Names ? (Y/N)	Y {for Yes}
New Seg name ?	SegMain <ret>

SUMMARY OF THE SEGMENT LOADER

```
PROCEDURE UnloadSeg    (routineAddr: Ptr);
PROCEDURE GetAppParms  (VAR apName: Str255; VAR apRefNum: INTEGER;
                        VAR apParam: Handle);
```

Chain {register-based}

Input: A0 points to application's file name pointer followed by a word telling which sound and video buffers to use.

Output: The application parameters for standard input and output.

```
PROCEDURE ExitToShell;
```

Advanced Routines

Launch {register-based}

Input: A0 points to application's file name pointer followed by a word telling which sound and video buffers to use.

Output: The application parameters--standard input and output and the Finder information handle.

```
PROCEDURE LoadSeg (segID: INTEGER);
```

GLOSSARY

application parameters: Information stored in 32 bytes of memory just above the application globals when an application is started up.

jump table: A table that contains one entry for every routine in an application and is the means by which the loading and unloading of segments is implemented.

main segment: The segment containing the main program.

segment: One of several parts into which the code of an application may be divided. Not all segments need to be in memory at the same time.

9-March-83
LAK

The OS Event Manager

The Event Manager core routines manipulate events on the system event queue. These consist of functions such as adding and retrieving events from the system event queue, polling for available events, and removing events from the queue. The system queue is initialized to contain 30 22-byte elements.

(ToolEvents contain the higher-level ToolBox event handling calls EventAvail and GetNextEvent: these will be documented separately with other ToolBox documentation, although some ToolEvents-defined events are briefly covered here. ToolEvents makes calls to OSEventAvail and GetOSEvent, adding Activate and Update events, and supports journaling. Most application programs will just make calls to ToolEvents.)

Four routines are associated with the event manager: PostEvent, OSEventAvail, GetOSEvent, and FlushEvents. PostEvent may be called from an interrupt or completion routine; all other routines in the event manager must be called from the main thread of execution. Additionally, the system event mask may be read and set via the OS routines GetSysParam and SetSysParam.

The Event Manager manages its own private buffer to get storage for the event queueing elements. It does this because PostEvent runs at interrupt level and thus cannot call the standard storage allocater.

Events

The Macintosh operating system uses the metaphor of an "event" to report to user programs the occurance of keyboard keypresses, mouse button state changes, and other relatively slow and irregular things which the system detects and the user program is interested in. Faster input/output, such as receipt of a character on one of the serial port , is handled via the "I/O driver" model in the I/O and File subsystems.

Event Mask, Event Number

Events are posted and selected subject to event masks; an event mask is a word-long bitmap of all possible events: a 1 in the bit position of an event enables that event. Possible events by event number, bit position in event mask, and name are:

0	$0001	Null Event
1	$0002	Mouse button down
2	$0004	Mouse button up
3	$0008	Key down
4	$0010	Key up
5	$0020	Auto-key
6	$0040	Update event
7	$0080	Disk Inserted

8	$0100	Activate/Deactivate event
9	$0200	Abort event
10	$0400	Network event
11	$0800	IO Driver event
12	$1000	application defined
13	$2000	application defined
14	$4000	application defined
15	$8000	application defined

Event Queue Element, Event Record

The basic data structure for events is a 22-byte buffer called an EVENT QUEUE ELEMENT, in which events are buffered by the Event Manager. Events are communicated to users via EVENT RECORDS, which are structured like event queue elements, minus the six-byte queue link and type fields. The SYSTEM EVENT BUFFER has room enough for 30 event queue elements.

Event Queue Element:

(0) Queue link to next element, zero for last element (32-bit)
(4) Queue type field, set to $0004 (16-bit)
(6) Event Record (16-byte)

Event Record:

(0) Event Number (16-bit)
(2) Event-defined message (32-bit)
(6) TICKS value when event occurred (32-bit) (TICKS is a 32-bit variable which is incremented every 1/60 second)
(10) Mouse position when event occurred (32-bit)
(14) Meta-key flags (8-bit) as follows (bit=1 when key is down):
bit 7-4: undefined
3: option key
2: alpha-lock key
1: shift key
0: command
(15) Mouse button state (8-bit):
bit 7: down=0,up=1
6-0: undefined (toolevents uses bits 0-1 to distinguish activate from deactivate, and sys-appl change).

Event-defined messages are as follows (including ToolEvents-defined events):

Null Event	none (0)
Mouse button down	none (0)
Mouse button up	none (0)
Key down	byte0=byte1=0,byte2=raw keycode,byte3=ASCII code
Key up	byte0=byte1=0,byte2=raw keycode,byte3=ASCII code
Auto-key	byte0=byte1=0,byte2=raw keycode,byte3=ASCII code
Disk Inserted	drive number: 1 internal, 2 external
Update event	32-bit windowPtr of window to be updated
Activate/Deactivate	32-bit windowPtr

Events are generally posted as they occur and are self-explanatory;

MACINTOSH USER EDUCATION

The File Manager: A Programmer's Guide /OS/FS

See Also: The Macintosh User Interface Guidelines
The Memory Manager: A Programmer's Guide
Inside Macintosh: A Road Map
Macintosh Packages: A Programmer's Guide
The Structure of a Macintosh Application
Programming Macintosh Applications in Assembly Language

Modification History: First Draft (ROM 7) Bradley Hacker 5/21/84

ABSTRACT

This manual describes the File Manager, the part of the Macintosh Operating System that controls the exchange of information between a Macintosh application and files.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the File Manager, the part of the Macintosh Operating System that controls the exchange of information between a Macintosh application and files. *** Eventually it will become part of the comprehensive Inside Macintosh manual. *** The File Manager allows you to create and access any number of files containing whatever information you choose.

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal. You should also be familiar with the following:

- the basic concepts behind the Macintosh Operating System's Memory Manager
- devices and device drivers, as described in the Inside Macintosh Road Map

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly-language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it.

The manual begins with an introduction to the File Manager and what you can do with it. It then discusses some basic concepts behind the File Manager: what files and volumes are and how they're accessed.

A section on using the File Manager introduces its routines and tells how they fit into the flow of your application. This is followed by sections explaining the File Manager's simplest, "high-level" Pascal routines and then its more complex, "low-level" Pascal and assembly-language routines. Both sections give detailed descriptions of all the procedures and functions, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that won't interest all readers. The data structures that the File Manager uses to store information in memory and on disks are described, and special information is provided for programmers who want to write their own file system.

Finally, there's a summary of the File Manager, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE FILE MANAGER

The File Manager is the part of the Operating System that handles communication between an application and files on block devices such as disk drives. Files are a principal means by which data is stored and transmitted on the Macintosh. A file is a named, ordered sequence of

bytes. The File Manager contains routines used to read and write to files.

Volumes

A volume is a piece of storage medium, such as a disk, formatted to contain files. A volume can be an entire disk or only part of a disk. Currently, the 3 1/2-inch Macintosh disks are one volume.

(note)

Specialized memory devices other than disks can also contain volumes, but the information in this manual applies only to volumes on disks.

You identify a volume by its volume name, which consists of any sequence of 1 to 27 printing characters. Volume names must always be followed by a colon (:) to distinguish them from other names. You can use uppercase and lowercase letters when naming volumes, but the File Manager ignores case when comparing names (it doesn't ignore diacritical marks).

(note)

The colon (:) after a volume name should only be used when calling File Manager routines; it should never be seen by the user.

A volume contains descriptive information about itself, including its name and a file directory listing information about files contained on the volume; it also contains files. The files are contained in allocation blocks, which are areas of volume space occupying multiples of 512 bytes.

A volume can be mounted or unmounted. A volume becomes mounted when it's in a disk drive and the File Manager reads descriptive information about the volume into memory. Once mounted, a volume may remain in a drive or be ejected. Only mounted volumes are known to the File Manager, and an application can access information on mounted volumes only. A volume becomes unmounted when the File Manager releases the memory used to store the descriptive information. Your application should unmount a volume when it's finished with the volume, or when it needs the memory occupied by the volume.

The File Manager assigns each mounted volume a volume reference number that you can use instead of its volume name to refer to it. Every mounted volume is also assigned a volume buffer, which is temporary storage space on the heap used when reading and writing information on the volume. The number of volumes that may be mounted at any time is limited only by the number of drives attached and available memory.

A mounted volume can be on-line or off-line. A mounted volume is on-line as long as the volume buffer and all the descriptive information read from the volume when it was mounted remain in memory (about 1K to 1.5K bytes); it becomes off-line when all but 94 bytes of

descriptive information are released. You can access information on on-line volumes immediately, but off-line volumes must be placed on-line before their information can be accessed. An application should place a volume off-line whenever it needs most of the memory the volume occupies. When an application ejects a volume from a drive, the File Manager automatically places the volume off-line.

To prevent unauthorized writing to a volume, volumes can be locked. Locking a volume involves either setting a software flag on the volume or changing some part of the volume physically (for example, sliding a tab from one position to another on a disk). Locking a volume ensures that none of the data on the volume can be changed.

Accessing Volumes

You can access a mounted volume via its volume name or volume reference number. On-line volumes in disk drives can also be accessed via the drive number of the drive on which the volume is mounted (the internal drive is number 1, the external drive is number 2, and any additional drives connected via a serial port will have larger numbers). When accessing a mounted volume, you should always use the volume name or volume reference number, rather than a drive number, because the volume may have been ejected or placed off-line. Whenever possible, use the volume reference number (to avoid confusion between volumes with the same name).

One volume is always the default volume. Whenever you call a routine to access a volume but don't specify which volume, the default volume is accessed. Initially, the volume used to start up the system is the default volume, but an application can designate any mounted volume as the default volume.

Whenever the File Manager needs to access a mounted volume that's been ejected from its drive, the dialog box shown in Figure 1 is displayed, and the File Manager waits until the user inserts the volume named volName into a drive.

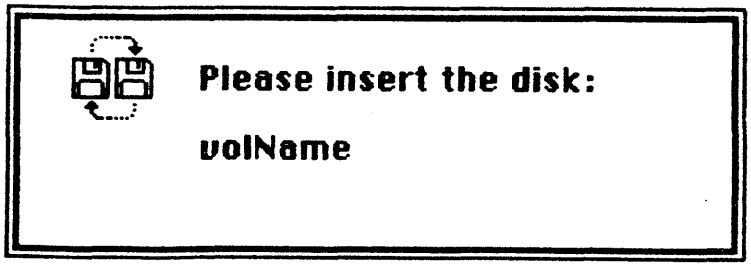


Figure 1. Disk-Switch Dialog

Files

A file is a finite sequence of numbered bytes. Any byte or group of bytes in the sequence can be accessed individually. A file is identified by its file name and version number. A file name consists of any sequence of 1 to 255 printing characters, excluding colons (:). You can use uppercase and lowercase letters when naming volumes, but the File Manager ignores case when comparing names (it doesn't ignore diacritical marks). The version number is any number from Ø to 255, and is used by the File Manager to distinguish between different files with the same name. A byte within a file is identified by its position within the ordered sequence.

(warning)

Your application should constrain file names to fewer than 64 characters, because the Finder will generate an error if given a longer name. You should always assign files a version number of Ø, because the Resource Manager and Segment Loader won't operate on files with nonzero file numbers, the Finder ignores version numbers, and the Standard File Package clears version numbers.

There are two parts or forks to a file: the data fork and the resource fork. Normally the resource fork of an application file contains the resources used by the application such as menus, fonts, and icons, and also the application code itself. The data fork can contain anything an application wants to store there. Information stored in resource forks should always be accessed via the Resource Manager. Information in data forks can only be accessed via the File Manager. For simplicity, "file" will be used instead of "data fork" in this manual.

A file can contain anywhere from Ø to 16,777,216 bytes (16 megabytes). Each byte is numbered: the first byte is byte Ø. You can read bytes from and write bytes to a file either singly or in sequences of unlimited length. Each read or write operation can start anywhere in the file, regardless of where the last operation began or ended. Figure 2 shows the structure of a file.

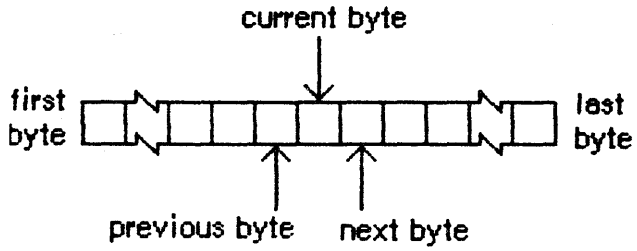


Figure 2. A File

A file's maximum size is defined by its physical end-of-file, which is 1 greater than the number of the last byte in its last allocation block (Figure 3). The physical end-of-file is equivalent to the maximum

number of bytes the file can contain. A file's actual size is defined by its logical end-of-file, which is 1 greater than the number of the last byte in the file. The logical end-of-file is equivalent to the actual number of bytes in the file, since the first byte is byte number Ø. The physical end-of-file is always greater than the logical end-of-file. For example, an empty file (one with Ø bytes) in a 1K-byte allocation block has a logical end-of-file of Ø and a physical end-of-file of 1Ø24. A file with 5Ø bytes has a logical end-of-file of 5Ø and a physical end-of-file of 1Ø24.

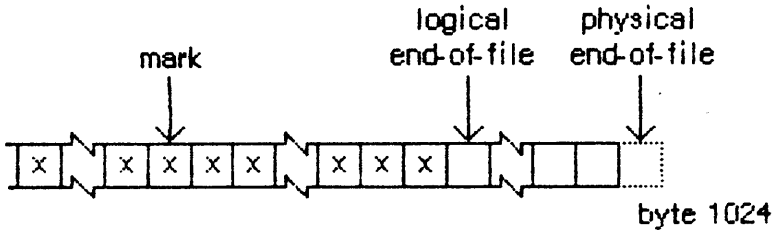


Figure 3. End-of-File and Mark

The current position marker, or mark, is the number of the next byte that will be read or written. The value of the mark can't exceed the value of the logical end-of-file. The mark automatically moves forward one byte for every byte read from or written to the file. If, during a write operation, the mark meets the logical end-of-file, both are moved forward one position for every additional byte written to the file. Figure 4 shows the movement of the mark and logical end-of-file.

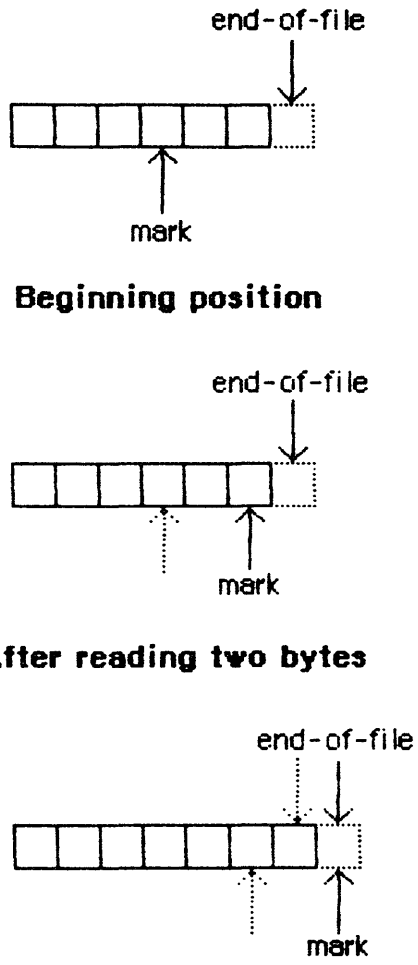


Figure 4. Movement of Logical End-of-File and Mark

If, during a write operation, the mark must move past the physical end-of-file, another allocation block is added to the file--the physical end-of-file is placed one byte beyond the end of the new allocation block, and the mark and logical end-of-file are placed at the first byte of the new allocation block.

An application can move the logical end-of-file to anywhere from the beginning of the file to the physical end-of-file (the mark is adjusted accordingly). If the logical end-of-file is moved to a position more than one allocation block short of the current physical end-of-file, the unneeded allocation block will be deleted from the file. The mark can be placed anywhere from the first byte in the file to the logical end-of-file.

Accessing Files

A file can be open or closed. An application can only perform certain operations, such as reading and writing, on open files; other operations, such as deleting, can only be performed on closed files.

To open a file, you must identify the file and the volume containing it. When a file is opened, the File Manager creates an access path, a description of the route to be followed when accessing the file. The access path specifies the volume on which the file is located (by volume reference number, drive number, or volume name) and the location of the file on the volume. Every access path is assigned a unique path reference number used to refer to it. You should always refer to a file via its path reference number, so that files with the same name aren't confused with one another.

A file can have one access path open for writing or for both reading and writing, and one or more access paths for reading only; there cannot be more than one access path that writes to a file. Each access path is separate from all other access paths to the file. A maximum of 12 access paths can be open at one time. Each access path can move its own mark and read at the position it indicates. All access paths to the same file share common logical and physical end-of-file markers.

The File Manager reads descriptive information about a newly opened file from its volume and stores it in memory. For example, each file has open permission information, which indicates whether data can only be read from it, or both read from and written to it. Each access path contains read/write permission information that specifies whether data is allowed to be read from the file, written to the file, both read and written, or whatever the file's open permission allows. If an application wants to write data to a file, both types of permission information must allow writing; if either type allows reading only, then no data can be written.

When an application requests that data be read from a file, the File Manager reads the data from the file and transfers it to the application's data buffer. Any part of the data that can be transferred in entire 512-byte blocks is transferred directly. Any part of the data composed of fewer than 512 bytes is also read from the file in one 512-byte block, but placed in temporary storage space in memory. Then, only the bytes containing the requested data are transferred to the application.

When an application writes data to a file, the File Manager transfers the data from the application's data buffer and writes it to the file. Any part of the data that can be transferred in entire 512-byte blocks is written directly. Any part of the data composed of fewer than 512 bytes is placed in temporary storage space in memory until 512 bytes have accumulated; then the entire block is written all at once.

Normally the temporary space in memory used for all reading and writing is the volume buffer, but an application can specify that an access path buffer be used instead for a particular access path (Figure 5).

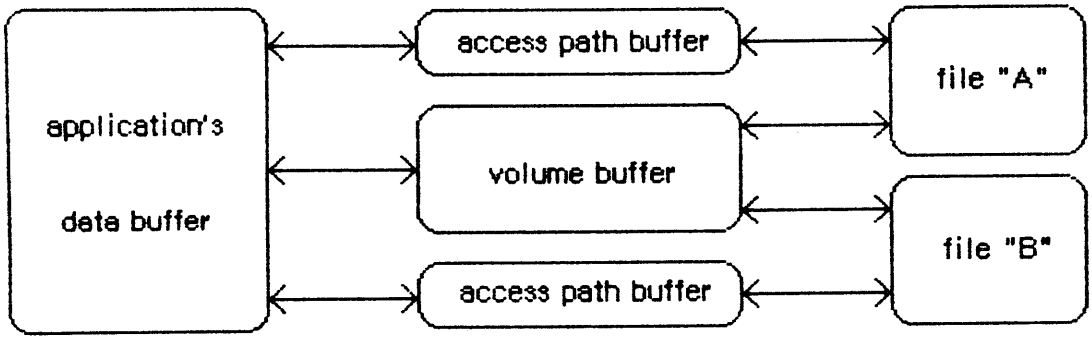


Figure 5. Buffers For Transferring Data

(warning)

You must lock every access path buffer you use, so its location doesn't change while the file is open.

Your application can lock a file to prevent unauthorized writing to it. Locking a file ensures that none of the data in it can be changed *** Currently, the Finder won't let you rename or delete a locked file, but it will let you change the data the file contains ***.

(note)

Advanced programmers: The File Manager can also read a continuous stream of characters or a line of characters. In the first case, you ask the File Manager to read a specific number of bytes: when that many have been read or when the mark has reached the logical end-of-file, the read operation terminates. In the second case, called newline mode, the read will terminate when either of the above conditions is fulfilled or when a specified character, the newline character, is read. The newline character is usually Return (ASCII code $ØD), but can be any character whose ASCII code is between $ØØ and $FF, inclusive. Information about newline mode is associated with each access path to a file, and can differ from one access path to another.

FILE INFORMATION USED BY THE FINDER

A file directory on a volume lists information about all the files on the volume. The information used by the Finder is contained in a data structure of type FInfo:

```
TYPE FInfo = RECORD
               fdType:     OSType;  {type of file}
               fdCreator:  OSType;  {file's creator}
               fdFlags:    INTEGER; {flags}
               fdLocation: Point;   {file's location}
               fdFldr:     INTEGER  {file's window}
             END;
```

Normally an application need only set the file type and creator when a file is created, and the Finder will manipulate the other fields. (File type and creator are discussed in The Structure of a Macintosh Application.) Advanced programmers may be interested in changing the contents of the other fields as well.

FdFlags indicates whether the file's icon is invisible, whether the file has a bundle, and other characteristics used internally by the Finder:

Bit	Meaning if set
5	File has a bundle
6	File's icon is invisible

Masks for these two bits are available as predefined constants:

```
CONST fHasBundle = 32; {set if file has a bundle}
      fInvisible = 64; {set if file's icon is invisible}
```

When you first install an application, you'll need to set its "bundle bit", as described in The Structure of a Macintosh Application. Whenever you create a file with a bundle, you'll need to set its bundle bit.

The next two fields indicate where the file's icon will appear if the icon is visible. FdLocation contains the location of the file's icon in its window, given in the local coordinate system of the window. FdFldr indicates the window in which the file's icon will appear, and may contain one of the following predefined constants:

```
CONST fTrash   = -3; {file is in trash window}
      fDesktop = -2; {file is on desktop}
      fDisk    =  Ø; {file is in disk window}
```

If fdFldr contains a positive number, the file's icon will appear in a folder; the numbers that identify folders are assigned by the Finder. Advanced programmers can get the folder number of an existing file, and place additional files in that same folder.

USING THE FILE MANAGER

This section discusses how the File Manager routines fit into the general flow of an application program and gives an idea of what routines you'll need to use. The routines themselves are described in

detail in the next two sections.

You can call File Manager routines via three different methods: high-level Pascal calls, low-level Pascal calls, and assembly language. The high-level Pascal calls are designed for Pascal programmers interested in using the File Manager in a simple manner; they provide adequate file I/O and don't require much special knowledge to use. The low-level Pascal and assembly-language calls are designed for advanced Pascal programmers and assembly-language programmers interested in using the File Manager to its fullest capacity; they require some special knowledge to be used most effectively.

Information for all programmers follows here. The next two sections contain special information for high-level Pascal programmers and for low-level Pascal and assembly-language programmers.

(note)

The names used to refer to routines here are actually the assembly-language macro names for the low-level routines, but the Pascal routine names are very similar.

The File Manager is automatically initialized each time the system is started up.

To create a new, empty file, call Create. Create allows you to set some of the information stored on the volume about the file.

To open a file, call Open. The File Manager creates an access path and returns a path reference number that you'll use every time you want to refer to it. Before you open a file, you may want to call the Standard File Package, which presents the standard interface through which the user can specify the file to be opened. The Standard File Package will return the name of the file, the volume reference number of the volume containing the file, and additional information. (If the user inserts an unmounted volume into a drive, the Standard File Package will automatically call the Disk Initialization Package to attempt to mount it.)

After opening a file, you can transfer data from it to an application's data buffer with Read, and send data from an application's data buffer to the file with Write. Read and Write allow you to specify a byte position within the data buffer, a number of bytes to transfer, and the location within the file. You can't use Write on a file whose open permission only allows reading, or on a file on a locked volume.

Once you've completed whatever reading and writing you want to do, call Close to close the file. Close writes the contents of the file's access path buffer to the volume and deletes the access path. You can remove a closed file (both forks) from a volume by calling Delete.

To protect against power loss or unexpected disk ejection, you should periodically call FlushVol (probably after each time you close a file), which writes the contents of the volume buffer and all access path buffers (if any) to the volume and updates the descriptive information

contained on the volume.

Whenever your application is finished with a disk, or the user chooses Eject from a menu, call Eject. Eject calls FlushVol, places the volume off-line, and then physically ejects the volume from its drive.

The preceding paragraphs covered the simplest File Manager routines: Open, Read, Write, Close, FlushVol, Eject, and Create. The remainder of this section describes the less commonly used routines, some of which are available only to advanced programmers. Skip the remainder of this section if the preceding paragraphs have provided you with all the information you want to know about using the File Manager.

When the Toolbox Event Manager function GetNextEvent receives a disk-inserted event, it calls the Desk Manager function SystemEvent. SystemEvent calls the File Manager function MountVol, which attempts to mount the volume on the disk. GetNextEvent then returns the disk-inserted event: the low-order word of the event message contains the number of the drive, and the high-order word contains the result code of the attempted mounting. If the result code indicates that an error occurred, you'll need to call the Disk Initialization Package to allow the user to initialize or eject the volume.

(note)

> Applications that rely on the Operating System Event Manager function GetOSEvent to learn about events (and don't call GetNextEvent) must explicitly call MountVol to mount volumes.

After a volume has been mounted, your application can call GetVolInfo, which will return the name of the volume, the amount of unused space on the volume, and a volume reference number that you can use every time you refer to that volume.

To minimize the amount of memory used by mounted volumes, an application can unmount or place off-line any volumes that aren't currently being used. To unmount a volume, call UnmountVol, which flushes a volume (by calling FlushVol) and releases all of the memory used for it (releasing about 1 to 1.5K bytes). To place a volume off-line, call OffLine, which flushes a volume (by calling FlushVol) and releases all of the memory used for it except for 94 bytes of descriptive information about the volume. Off-line volumes are placed on-line by the File Manager as needed, but your application must remount any unmounted volumes it wants to access. The File Manager itself may place volumes off-line during its normal operation.

If you would like all File Manager calls to apply to one volume, you can specify that volume as the default. You can use SetVol to set the default volume to any mounted volume, and GetVol to learn the name and volume reference number of the default volume.

Normally, volume initialization and naming is handled by the Standard File Package, which calls the Disk Initialization Package. If you want to initialize a volume explicitly or erase all files from a volume, you

can call the Disk Initialization Package directly. When you want to change the name of a volume, call the File Manager function Rename.

Applications normally will use the Resource Manager to open resource forks and change the information contained within, but programmers writing unusual applications (such as a disk-copying utility) might want to use the File Manager to open resource forks. This is done by calling OpenRF. As with Open, the File Manager creates an access path and returns a path reference number that you'll use every time you want to refer to this resource fork.

As an alternative to specifying byte positions within a file with Read and Write, you can specify the byte position of the mark by calling SetFPos. GetFPos returns the byte position of the mark.

Whenever a disk has been reconstructed in an attempt to salvage lost files (because its directory or other file-access information has been destroyed), the logical end-of-file of each file will probably be equal to each physical end-of-file, regardless of where the actual logical end-of-file is. The first time an application attempts to read from a file on a reconstructed volume, it will blindly pass the correct logical end-of-file and read misinformation until it reaches the new, incorrect logical end-of-file. To prevent this from occurring, an application should always maintain an independent record of the logical end-of-file of each file it uses. To determine the File Manager's conception of the length of a file, or find out how many bytes have yet to be read from it, call GetEOF, which returns the logical end-of-file. You can change the length of a file by calling SetEOF.

Allocation blocks are automatically added to and deleted from a file as necessary. If this happens to a number of files alternately, each of the files will be contained in allocation blocks scattered throughout the volume, which increases the time required to access those files. To prevent such fragmentation of files, you can allocate a number of contiguous allocation blocks to an open file by calling Allocate.

Instead of calling FlushVol, an unusual application might call FlushFile. FlushFile forces the contents of a file's volume buffer and access path buffer (if any) to be written to its volume. FlushFile doesn't update the descriptive information contained on the volume, so the volume information won't be correct until you call FlushVol.

To get information about a file (such as its name and creation date) stored on a volume, call GetFileInfo. You can change this information by calling SetFileInfo. Changing the name or version number of a file is accomplished by calling Rename or SetFilType, respectively; they will have a similar effect, since both the file name and version number are needed to identify a file. You can lock or unlock a file by calling SetFilLock or RstFilLock, respectively.

You can't use Write, Allocate, or SetEOF on a locked file, a file whose open permission only allows reading, or a file on a locked volume. You can't use Rename or SetFilType on a file on a locked volume.

HIGH-LEVEL FILE MANAGER ROUTINES

This section describes all the high-level Pascal routines of the File Manager. Assembly-language programmers cannot call these routines. For information on calling the low-level Pascal and assembly-language routines, see the next section.

When accessing a volume, you must identify it by its volume name, its volume reference number, or the drive number of its drive--or allow the default volume to be accessed. The parameter names used in identifying a volume are volName, vRefNum, and drvNum. VRefNum and drvNum are both integers. VolName is a pointer, of type StringPtr, to a volume name.

The File Manager determines which volume to access by using one of the following:

1. VolName. (If volName points to a zero-length name, an error is returned.)

2. If volName is NIL or points to an improper volume name, then vRefNum or drvNum (only one is given per routine).

3. If vRefNum or drvNum is zero, the default volume. (If there isn't a default volume, an error is returned.)

(warning)
Before you pass a parameter of type StringPtr to a File Manager routine such as GetVol, be sure that memory has been allocated for the variable. For example, the following statements will ensure that memory is allocated for the variable myStr:

```
VAR myStr: Str255;
. . .
BEGIN
    result := GetVol(@myStr, myRefNum);
    . . .
END;
```

When accessing a closed file on a volume, you must identify the volume by the method given above, and identify the file by its name in the fileName parameter. (The high-level File Manager routines will work only with files having a version number of Ø.) FileName can contain either the file name alone or the file name prefixed by a volume name.

(note)
Although fileName can include both the volume name and the file name, applications shouldn't encourage users to prefix a file name with a volume name.

You cannot specify an access path buffer when calling high-level Pascal routines. All access paths open on a volume will share the volume buffer, causing a slight increase in the amount of time required to

access files.

All File Manager routines return a result code of type OSErr as their function result. Each routine description lists all of the applicable result codes, along with a short description of what the result code means. Lengthier explanations of all the result codes can be found in the summary at the end of this manual.

Accessing Volumes

```
FUNCTION GetVInfo (drvNum: INTEGER; volName: StringPtr; VAR vRefNum:
          INTEGER; VAR freeBytes: LongInt) : OSErr;
```

GetVInfo returns the name, reference number, and available space (in bytes), in volName, vRefNum, and freeBytes, for the volume in the specified drive.

Result codes		
	noErr	No error
	nsvErr	No default volume
	paramErr	Bad drive number

```
FUNCTION GetVol (volName: StringPtr; VAR vRefNum: INTEGER) : OSErr;
```

GetVol returns the name of the default volume in volName and its volume reference number in vRefNum.

Result codes		
	noErr	No error
	nsvErr	No default volume

```
FUNCTION SetVol (volName: StringPtr; vRefNum: INTEGER) : OSErr;
```

SetVol sets the default volume to the mounted volume specified by volName or vRefNum.

Result codes		
	noErr	No error
	bdNamErr	Bad volume name
	nsvErr	No such volume
	paramErr	No default volume

```
FUNCTION FlushVol (volName: StringPtr; vRefNum: INTEGER) : OSErr;
```

On the volume specified by volName or vRefNum, FlushVol writes the contents of the associated volume buffer and descriptive information about the volume (if they've changed since the last time FlushVol was called).

Result codes		
	noErr	No error
	bdNamErr	Bad volume name
	extFSErr	External file system
	ioErr	Disk I/O error
	nsDrvErr	No such drive
	nsvErr	No such volume
	paramErr	No default volume

```
FUNCTION UnmountVol (volName: StringPtr; vRefNum: INTEGER) : OSErr;
```

UnmountVol unmounts the volume specified by volName or vRefNum, by calling FlushVol to flush the volume buffer, closing all open files on the volume, and releasing the memory used for the volume.

(warning)
Don't unmount the startup volume.

Result codes		
	noErr	No error
	bdNamErr	Bad volume name
	extFSErr	External file system
	ioErr	Disk I/O error
	nsDrvErr	No such drive
	nsvErr	No such volume
	paramErr	No default volume

```
FUNCTION Eject (volName: StringPtr; vRefNum: INTEGER) : OSErr;
```

Eject calls FlushVol to flush the volume specified by volName or vRefNum, places the volume offline, and then ejects the volume.

Result codes		
	noErr	No error
	bdNamErr	Bad volume name
	extFSErr	External file system
	ioErr	Disk I/O error
	nsDrvErr	No such drive
	nsvErr	No such volume
	paramErr	No default volume

Changing File Contents

```
FUNCTION Create (fileName: Str255; vRefNum: INTEGER; creator: OSType;
          fileType: OSType) : OSErr;
```

Create creates a new file with the specified name, file type, and creator, on the specified volume. (File type and creator are discussed in The Structure of a Macintosh Application.) The new file is unlocked and empty. Its modification and creation dates are set to the time of the system clock.

Result codes		
	noErr	No error
	bdNamErr	Bad file name
	dupFNErr	Duplicate file name
	dirFulErr	Directory full
	extFSErr	External file system
	ioErr	Disk I/O error
	nsvErr	No such volume
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock

```
FUNCTION FSOpen (fileName: Str255; vRefNum: INTEGER; VAR refNum:
          INTEGER) : OSErr;
```

FSOpen creates an access path to the file having the name fileName on the specified volume. A path reference number is returned in refNum. The access path's read/write permission is set to whatever the file's open permission allows.

Result codes		
	noErr	No error
	bdNamErr	Bad file name
	extFSErr	External file system
	fnfErr	File not found
	ioErr	Disk I/O error
	mFulErr	Memory full
	nsvErr	No such volume
	opWrErr	File already open for writing
	tmfoErr	Too many files open

```
FUNCTION FSRead (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr) :
          OSErr;
```

FSRead attempts to read the number of bytes specified by the count parameter from the open file whose access path is specified by refNum, and transfer them to the data buffer pointed to by buffPtr. The read operation begins at the mark, so you might want to precede this with a call to SetFPos. If you try to read past the logical end-of-file, FSRead moves the mark to the end-of-file and returns eofErr as its function result. After the read is completed, the number of bytes actually read is returned in the count parameter.

Result codes		
	noErr	No error
	eofErr	End-of-file
	extFSErr	External file system
	fnOpnErr	File not open
	ioErr	Disk I/O error
	paramErr	Negative count
	rfNumErr	Bad reference number

```
FUNCTION FSWrite (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr) :
          OSErr;
```

FSWrite takes the number of bytes specified by the count parameter from the buffer pointed to by buffPtr and attempts to write them to the open file whose access path is specified by refNum. The write operation begins at the mark, so you might want to precede this with a call to SetFPos. After the write is completed, the number of bytes actually written is returned in the count parameter.

Result codes		
	noErr	No error
	dskFulErr	Disk full
	fLckdErr	File locked
	fnOpnErr	File not open
	ioErr	Disk I/O error
	paramErr	Negative count
	rfNumErr	Bad reference number
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock
	wrPermErr	Read/write or open permission doesn't allow writing

```
FUNCTION GetFPos (refNum: INTEGER; VAR filePos: LongInt) : OSErr;
```

GetFPos returns, in filePos, the mark of the open file whose access path is specified by refNum.

Result codes		
	noErr	No error
	extFSErr	External file system
	fnOpnErr	File not open
	ioErr	Disk I/O error
	rfNumErr	Bad reference number

```
FUNCTION SetFPos (refNum: INTEGER; posMode: INTEGER; posOff: LongInt) :
        OSErr;
```

SetFPos sets the mark of the open file whose access path is specified by refNum, to the position specified by posMode and posOff. PosMode indicates whether the mark should be set relative to the beginning of the file, the logical end-of-file, or the mark; it must contain one of the following predefined constants:

```
CONST fsAtMark     = 0; {at current position of mark }
                        { (posOff ignored)}
      fsFromStart  = 1; {offset relative to beginning of file}
      fsFromLEOF   = 2; {offset relative to logical end-of-file}
      fsFromMark   = 3; {offset relative to current mark}
```

PosOff specifies the byte offset (either positive or negative) relative to posMode where the mark should actually be set. If you try to set the mark past the logical end-of-file, SetFPos moves the mark to the end-of-file and returns eofErr as its function result.

Result codes		
	noErr	No error
	eofErr	End-of-file
	extFSErr	External file system
	fnOpnErr	File not open
	ioErr	Disk I/O error
	posErr	Tried to position before start of file
	rfNumErr	Bad reference number

```
FUNCTION GetEOF (refNum: INTEGER; VAR logEOF: LongInt) : OSErr;
```

GetEOF returns, in logEOF, the logical end-of-file of the open file whose access path is specified by refNum.

Result codes		
	noErr	No error
	extFSErr	External file system
	fnOpnErr	File not open
	ioErr	Disk I/O error
	rfNumErr	Bad reference number

```
FUNCTION SetEOF (refNum: INTEGER; logEOF: LongInt) : OSErr;
```

SetEOF sets the logical end-of-file of the open file whose access path is specified by refNum, to the position specified by logEOF. If you attempt to set the logical end-of-file beyond the physical end-of-file, the physical end-of-file is set to one byte beyond the end of the next free allocation block; if there isn't enough space on the volume, no change is made, and SetEOF returns dskFulErr as its function result. If logEOF is Ø, all space on the volume occupied by the file is released.

Result codes		
	noErr	No error
	dskFulErr	Disk full
	extFSErr	External file system
	fLckdErr	File locked
	fnOpnErr	File not open
	ioErr	Disk I/O error
	rfNumErr	Bad reference number
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock
	wrPermErr	Read/write or open permission doesn't allow writing

```
FUNCTION Allocate (refNum: INTEGER; VAR count: LongInt) : OSErr;
```

Allocate adds the number of bytes specified by the count parameter to the open file whose access path is specified by refNum, and sets the physical end-of-file to one byte beyond the last block allocated. The number of bytes allocated is always rounded up to the nearest multiple of the allocation block size, and returned in the count parameter. If there isn't enough empty space on the volume to satisfy the allocation request, the rest of the space on the volume is allocated, and Allocate returns dskFulErr as its function result.

Result codes		
	noErr	No error
	dskFulErr	Disk full
	fLckdErr	File locked
	fnOpnErr	File not open
	ioErr	Disk I/O error
	rfNumErr	Bad reference number
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock
	wrPermErr	Read/write or open permission doesn't allow writing

```
FUNCTION FSClose (refNum: INTEGER) : OSErr;
```

FSClose removes the access path specified by refNum, writes the contents of the volume buffer to the volume, and updates the file's entry in the file directory.

(note)
Some information stored on the volume won't be correct until FlushVol is called.

Result codes		
	noErr	No error
	extFSErr	External file system
	fnfErr	File not found
	fnOpnErr	File not open
	ioErr	Disk I/O error
	nsvErr	No such volume
	rfNumErr	Bad reference number

Changing Information About Files

All of the routines described in this section affect both forks of the file, and don't require the file to be open.

```
FUNCTION GetFInfo (fileName: Str255; vRefNum: INTEGER; VAR fndrInfo:
          FInfo) : OSErr;
```

For the file having the name fileName on the specified volume, GetFInfo returns information used by the Finder in fndrInfo (see the section "File Information Used by the Finder").

Result codes		
	noErr	No error
	bdNamErr	Bad file name
	extFSErr	External file system
	fnfErr	File not found
	ioErr	Disk I/O error
	nsvErr	No such volume
	paramErr	No default volume

```
FUNCTION SetFInfo (fileName: Str255; vRefNum: INTEGER; fndrInfo: FInfo)
          : OSErr;
```

For the file having the name fileName on the specified volume, SetFInfo sets information needed by the Finder to fndrInfo (see the section "File Information Used by the Finder").

Result codes		
	noErr	No error
	extFSErr	External file system
	fLckdErr	File locked
	fnfErr	File not found
	ioErr	Disk I/O error
	nsvErr	No such volume

	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock

```
FUNCTION SetFLock (fileName: Str255; vRefNum: INTEGER) : OSErr;
```

SetFLock locks the file having the name fileName on the specified volume. Access paths currently in use aren't affected.

Result codes	noErr	No error
	extFSErr	External file system
	fnfErr	File not found
	ioErr	Disk I/O error
	nsvErr	No such volume
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock

```
FUNCTION RstFLock (fileName: Str255; vRefNum: INTEGER) : OSErr;
```

RstFLock unlocks the file having the name fileName on the specified volume. Access paths currently in use aren't affected.

Result codes	noErr	No error
	extFSErr	External file system
	fnfErr	File not found
	ioErr	Disk I/O error
	nsvErr	No such volume
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock

```
FUNCTION Rename (oldName: Str255; vRefNum: INTEGER; newName: Str255) :
          OSErr;
```

Given a file name in oldName, Rename changes the name of the file to newName. Access paths currently in use aren't affected. Given a volume name in oldName or a volume reference number in vRefNum, Rename changes the name of the specified volume to newName.

Result codes	noErr	No error
	bdNamErr	Bad file name
	dirFulErr	Directory full
	dupFNErr	Duplicate file name
	extFSErr	External file system
	fLckdErr	File locked
	fnfErr	File not found
	fsRnErr	Renaming difficulty
	ioErr	Disk I/O error
	nsvErr	No such volume
	paramErr	No default volume
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock

```
FUNCTION FSDelete (fileName: Str255; vRefNum: INTEGER) : OSErr;
```

FSDelete removes the closed file having the name fileName from the specified volume.

(note)
This function will delete **both** forks of the file.

Result codes		
	noErr	No error
	bdNamErr	Bad file name
	extFSErr	External file system
	fBsyErr	File busy
	fLckdErr	File locked
	fnfErr	File not found
	ioErr	Disk I/O error
	nsvErr	No such volume
	vLckdErr	Software volume lock
	wPrErr	Hardware volume lock

LOW-LEVEL FILE MANAGER ROUTINES

This section contains special information for programmers using the low-level Pascal or assembly-language routines of the File Manager, and describes them in detail. For more information on using assembly language, see Programming Macintosh Applications in Assembly Language.

You can execute most File Manager routines either synchronously (meaning that the application must wait until the routine is completed) or asynchronously (meaning that the application is free to perform other tasks while the routine is executing). MountVol, UnmountVol, Eject, and OffLine cannot be executed asynchronously, because they use the Memory Manager to allocate and deallocate memory.

When an application calls a File Manager routine asynchronously, an I/O request is placed in the file I/O queue, and control returns to the calling application--even before the actual I/O is completed. Requests are taken from the queue one at a time (in the same order that they were entered), and processed. Only one request may be processed at any given time.

The calling application may specify a completion routine to be executed as soon as the I/O operation has been completed.

At any time, you can use the InitQueue procedure to clear all queued File Manager calls except the current one. InitQueue is especially useful when an error occurs and you no longer wish queued calls to be executed.

Routine parameters passed by an application to the File Manager and returned by the File Manager to an application are contained in a parameter block, which is memory space in the heap or stack. Most

low-level Pascal calls to the File Manager are of the form

```
PBCallName (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

PBCallName is the name of the routine. ParamBlock points to the parameter block containing the parameters for the routine. If async is TRUE, the call will be executed asynchronously; if FALSE, it will be executed synchronously. Each call returns an integer result code of type OSErr. Each routine description lists all of the applicable result codes, along with a short description of what the result code means. Lengthier explanations of all the result codes can be found in the summary at the end of this manual.

Assembly-language note: When you call a File Manager routine, AØ must point to a parameter block containing the parameters for the routine. If you want the routine to be executed asynchronously, set bit 1Ø of the routine trap word. You can do this by supplying the word ASYNC as the second argument to the routine macro. For example:

```
_Read  paramBlock,ASYNC
```

You can set or test bit 1Ø of a trap word by using the global constant asynTrpBit.

If you want a routine to be executed immediately (bypassing the file I/O queue), set bit 9 of the routine trap word. This can be accomplished by supplying the word IMMED as the second argument to the routine macro. For example:

```
_Write  paramBlock,IMMED
```

You can set or test bit 9 of a trap word by using the global constant noQueueBit. You can specify either ASYNC or IMMED, but not both.

All routines except InitQueue return a result code in DØ.

Routine Parameters

There are three different kinds of parameter blocks you'll pass to File Manager routines. Each kind is used with a particular set of routine calls: I/O routines, file information routines, and volume information routines.

The lengthy, variable-length data structure of a parameter block is given below. The Device Manager and File Manager use this same data structure, but only the parts relevant to the File Manager are shown

here. Each kind of parameter block contains eight fields of standard information and nine to 16 fields of additional information:

```
TYPE ParamBlkType = (ioParam, fileParam, volumeParam, cntrlParam);

     ParamBlockRec = RECORD
                       qLink:        QElemPtr;  {next queue entry}
                       qType:        INTEGER;   {queue type}
                       ioTrap:       INTEGER;   {routine trap}
                       ioCmdAddr:    Ptr;       {routine address}
                       ioCompletion: ProcPtr;   {completion routine}
                       ioResult:     OSErr;     {result code}
                       ioNamePtr:    StringPtr; {volume or file name}
                       ioVRefNum:    INTEGER;   {volume reference or }
                                                { drive number}
                       CASE ParamBlkType OF
                        ioParam:
                         . . .  {I/O routine parameters}
                        fileParam:
                         . . .  {file information routine parameters}
                        volumeParam:
                         . . .  {volume information routine parameters}
                        cntrlParam:
                         . . .  {Control and Status call parameters}
                     END;

     ParmBlkPtr = ^ParamBlockRec;
```

The first four fields in each parameter block are handled entirely by the File Manager, and most programmers needn't be concerned with them; programmers who are interested in them should see the section "Data Structures in Memory".

IOCompletion contains the address of a completion routine to be executed at the end of an asynchronous call; it should be NIL for asynchronous calls with no completion routine, and is automatically set to NIL for all synchronous calls. For asynchronous calls, ioResult is positive while the routine is executing, and returns the result code. Your application can poll ioResult during the asynchronous execution of a routine to determine when the routine has completed. Completion routines are executed after ioResult is returned.

IONamePtr points to either a volume name or a file name (which can be prefixed by a volume name).

(note)
Although ioNamePtr can include both the volume name and the file name, applications shouldn't encourage users to prefix a file name with a volume name.

IOVRefNum contains either the reference number of a volume or the drive number of a drive containing a volume.

For routines that access volumes, the File Manager determines which volume to access by using one of the following:

1. IONamePtr, a pointer to the volume name.

2. If ioNamePtr is NIL, or points to an improper volume name, then ioVRefNum. (If ioVRefNum is negative, it's a volume reference number; if positive, it's a drive number.)

3. If ioVRefNum is Ø, the default volume. (If there isn't a default volume, an error is returned.)

For routines that access closed files, the File Manager determines which file to access by using ioNamePtr, a pointer to the name of the file (and possibly also of the volume).

- If the string pointed to by ioNamePtr doesn't include the volume name, the File Manager uses steps 2 and 3 above to determine the volume.

- If ioNamePtr is NIL or points to an improper file name, an error is returned.

The first eight fields are adequate for a few calls, but most of the File Manager routines require more fields, as described below. The parameters used with Control and Status calls are described in the Device Manager manual *** doesn't yet exist ***.

I/O Parameters

When you call one of the I/O routines, you'll use these nine additional fields after the standard 8-field parameter block:

```
ioParam:
 (ioRefNum:    INTEGER;    {path reference number}
  ioVersNum:   SignedByte; {version number}
  ioPermssn:   SignedByte; {read/write permission}
  ioMisc:      Ptr;        {miscellaneous}
  ioBuffer:    Ptr;        {data buffer}
  ioReqCount:  LongInt;    {requested number of bytes}
  ioActCount:  LongInt;    {actual number of bytes}
  ioPosMode:   INTEGER;    {newline character and type of }
                           { positioning operation}
  ioPosOffset: LongInt);   {size of positioning offset}
```

For routines that access open files, the File Manager determines which file to access by using the path reference number in ioRefNum. IOPermssn requests permission to read or write via an access path, and must contain one of the following predefined constants:

```
CONST fsCurPerm  = Ø; {whatever is currently allowed}
      fsRdPerm   = 1; {request to read only
      fsWrPerm   = 2; {request to write only}
      fsRdWrPerm = 3; {request to read and write}
```

This request is compared with the open permission of the file. If the open permission doesn't allow I/O as requested, an error will be returned.

The content of ioMisc depends on the routine called; it contains either a pointer to an access path buffer, a new logical end-of-file, a new version number, or a pointer to a new volume or file name. Since ioMisc is of type Ptr, while end-of-file is LongInt and version number is SignedByte, you'll need to perform type conversions to correctly interpret the value of ioMisc.

IOBuffer points to a data buffer into which data is written by Read calls and from which data is read by Write calls. IOReqCount specifies the requested number of bytes to be read, written, or allocated. IOActCount contains the number of bytes actually read, written, or allocated.

IOPosMode and ioPosOffset contain positioning information used for Read, Write, and SetFPos calls. Bits Ø and 1 of ioPosMode indicate how to position the mark, and you can use the following predefined constants to set or test their value:

```
CONST fsAtMark    = Ø; {at current position of mark }
                       { (ioPosOffset ignored)}
      fsFromStart = 1; {offset relative to beginning of file}
      fsFromLEOF  = 2; {offset relative to logical end-of-file}
      fsFromMark  = 3; {offset relative to current mark}
```

IOPosOffset specifies the byte offset (either positive or negative) relative to ioPosMode where the operation will be performed.

Assembly-language note: If bit 6 of ioPosMode is set, the File Manager will verify that all data read into memory by a Read call exactly matches the data on the volume (ioErr will be returned if any of the data doesn't match).

(note)

Advanced programmers: Bit 7 of ioPosMode is the newline flag--set if read operations should terminate at newline characters, and clear if reading should terminate at the end of the access path buffer or volume buffer. The high-order byte of ioPosMode contains the ASCII code of the newline character.

File Information Parameters

When you call the PBGetFileInfo and PBSetFileInfo functions, you'll use the following 16 additional fields after the standard 8-field parameter block:

```
fileParam:
 (ioFRefNum:     INTEGER;     {path reference number}
  ioFVersNum:    SignedByte;  {version number}
  filler1:       SignedByte;  {not used}
  ioFDirIndex:   INTEGER;     {file number}
  ioFlAttrib:    SignedByte;  {file attributes}
  ioFlVersNum:   SignedByte;  {version number}
  ioFlFndrInfo:  FInfo;       {information used by the Finder}
  ioFlNum:       LongInt;     {file number}
  ioFlStBlk:     INTEGER;     {first allocation block of data fork}
  ioFlLgLen:     LongInt;     {logical end-of-file of data fork}
  ioFlPyLen:     LongInt;     {physical end-of-file of data fork}
  ioFlRStBlk:    INTEGER;     {first allocation block of resource fork}
  ioFlRLgLen:    LongInt;     {logical end-of-file of resource fork}
  ioFlRPyLen:    LongInt;     {physical end-of-file of resource fork}
  ioFlCrDat:     LongInt;     {date and time of creation}
  ioFlMdDat:     LongInt);    {date and time of last modification}
```

IOFDirIndex contains the file number, another method of referring to a file; most programmers needn't be concerned with file numbers, but those interested can read the section "Data Organization on Volumes".

Assembly-language note: IOFlAttrib contains eight bits of file attributes: if bit 7 is set, the file is open; if bit Ø is set, the file is locked.

IOFlStBlk and ioFlRStBlk contain Ø if the file's data or resource fork is empty, respectively. The date and time in the ioFlCrDat and ioFlMdDat fields are specified in seconds since 12:ØØ AM, January 1, 19Ø4.

Volume Information Parameters

When you call GetVolInfo, you'll use the following 14 additional fields:

```
volumeParam:
 (filler2:     LongInt;  {not used}
  ioVolIndex:  INTEGER;  {volume index}
  ioVCrDate:   LongInt;  {date and time of initialization}
  ioVLsBkUp:   LongInt;  {date and time of last volume backup}
  ioVAtrb:     INTEGER;  {bit 15=1 if volume locked}
  ioVNmFls:    INTEGER;  {number of files in file directory}
  ioVDirSt:    INTEGER;  {first block of file directory}
  ioVBlLn:     INTEGER;  {number of blocks in file directory}
  ioVNmAlBlks: INTEGER;  {number of allocation blocks on volume}
  ioVAlBlkSiz: LongInt;  {number of bytes per allocation block}
  ioVClpSiz:   LongInt;  {number of bytes to allocate}
  ioAlBlSt:    INTEGER;  {first block in volume block map}
  ioVNxtFNum:  LongInt;  {next free file number}
  ioVFrBlk:    INTEGER); {number of free allocation blocks}
```

IOVolIndex contains the volume index, another method of referring to a volume; the first volume mounted has an index of 1, and so on. Most programmers needn't be concerned with the parameters providing information about file directories and block maps (such as ioVNmFls), but interested programmers can read the section "Data Organization on Volumes".

Routine Descriptions

This section describes the procedures and functions. Each routine description includes the low-level Pascal form of the call and the routine's assembly-language macro. A list of the fields in the parameter block affected by the call is also given.

Assembly-language note: The field names given in these descriptions are those of the ParamBlockRec data type; see the "Summary of the File Manager" for the equivalent assembly-language equates.

The number next to each parameter name indicates the byte offset of the parameter from the start of the parameter block pointed to by A0; only assembly-language programmers need be concerned with it. An arrow drawn next to each parameter name indicates whether it's an input, output, or input/output parameter:

Arrow	Meaning
-->	Parameter must be passed to the routine
<--	Parameter will be returned by the routine
<-->	Parameter must be passed to and will be returned by the routine

Initializing the File I/O Queue

```
PROCEDURE InitQueue;
```

Trap macro _InitQueue

InitQueue clears all queued File Manager calls except the current one. There are no parameters or result codes associated with InitQueue.

Accessing Volumes

```
FUNCTION PBMountVol (paramBlock: ParmBlkPtr) : OSErr;
```

Trap macro _MountVol

Parameter block

←–	16	ioResult	word
←→	22	ioVRefNum	word

Result codes

noErr	No error
badMDBErr	Master directory block is bad
extFSErr	External file system
ioErr	Disk I/O error
mFulErr	Memory full
noMacDskErr	Not a Macintosh volume
nsDrvErr	No such drive
paramErr	Bad drive number
volOnLinErr	Volume already on-line

PBMountVol mounts the volume in the drive whose number is ioVRefNum, and returns a volume reference number in ioVRefNum. If there are no volumes already mounted, this volume becomes the default volume. PBMountVol is always executed synchronously.

```
FUNCTION PBGetVolInfo (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _GetVolInfo

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
<->	18	ioNamePtr	pointer
<->	22	ioVRefNum	word
-->	28	ioVolIndex	word
<--	30	ioVCrDate	long word
<--	34	ioVLsBkUp	long word
<--	38	ioVAtrb	word
<--	40	ioVNmFls	word
<--	42	ioVDirSt	word
<--	44	ioVBlLn	word
<--	46	ioVNmAlBlks	word
<--	48	ioVAlBlkSiz	long word
<--	52	ioVClpSiz	long word
<--	56	ioAlBlSt	word
<--	58	ioVNxtFNum	long word
<--	62	ioVFrBlk	word

Result codes

noErr	No error
nsvErr	No such volume
paramErr	No default volume

PBGetVolInfo returns information about the specified volume. If ioVolIndex is positive, the File Manager attempts to use it to find the volume. If ioVolIndex is negative, the File Manager uses ioNamePtr and ioVRefNum in the standard way to determine which volume. If ioVolIndex is 0, the File Manager attempts to access the volume by using ioVRefNum only. The volume reference number is returned in ioVRefNum, and the volume name is returned in ioNamePtr (unless ioNamePtr is NIL).

```
FUNCTION PBGetVol (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _GetVol

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
<--	18	ioNamePtr	pointer
<--	22	ioVRefNum	word

Result codes

noErr	No error
nsvErr	No default volume

PBGetVol returns the name of the default volume in ioNamePtr and its volume reference number in ioVRefNum (unless ioNamePtr is NIL).

```
FUNCTION PBSetVol (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _SetVol

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word

Result codes

noErr	No error
bdNamErr	Bad volume name
nsvErr	No such volume
paramErr	No default volume

PBSetVol sets the default volume to the mounted volume specified by ioNamePtr or ioVRefNum.

```
FUNCTION PBFlshVol (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _FlushVol

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word

Result codes

noErr	No error
bdNamErr	Bad volume name
extFSErr	External file system
ioErr	Disk I/O error
nsDrvErr	No such drive
nsvErr	No such volume
paramErr	No default volume

PBFlshVol writes descriptive information, the contents of the associated volume buffer, and all access path buffers to the volume specified by ioNamePtr or ioVRefNum, to the volume (if they've changed since the last time PBFlshVol was called). The volume modification date is set to the current time.

FUNCTION PBUnmountVol (paramBlock: ParmBlkPtr) : OSErr;

Trap macro _UnmountVol

Parameter block

<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word

Result codes

noErr	No error
bdNamErr	Bad volume name
extFSErr	External file system
ioErr	Disk I/O error
nsDrvErr	No such drive
nsvErr	No such volume
paramErr	No default volume

PBUnmountVol unmounts the volume specified by ioNamePtr or ioVRefNum, by calling PBFlshVol to flush the volume, closing all open files on the volume, and releasing all the memory used for the volume. PBUnmountVol is always executed synchronously.

(warning)
Don't unmount the startup volume.

FUNCTION PBOffLine (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;

Trap macro _OffLine

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word

Result codes

noErr	No error
bdNamErr	Bad volume name
extFSErr	External file system
ioErr	Disk I/O error
nsDrvErr	No such drive
nsvErr	No such volume
paramErr	No default volume

PBOffLine places off-line the volume specified by ioNamePtr or ioVRefNum, by calling PBFlshVol to flush the volume, and releasing all the memory used for the volume except for 94 bytes of descriptive information.

```
FUNCTION PBEject (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _Eject

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word

Result codes

noErr	No error
bdNamErr	Bad volume name
extFSErr	External file system
ioErr	Disk I/O error
nsDrvErr	No such drive
nsvErr	No such volume
paramErr	No default volume

PBEject calls PBOffLine to place the volume specified by ioNamePtr or ioVRefNum off-line, and then ejects the volume.

You may call PBEject asynchronously; the first part of the call is executed synchronously, and the actual ejection is executed asynchronously.

Changing File Contents

```
FUNCTION PBCreate (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _Create

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
-->	26	ioVersNum	byte

Result codes

noErr	No error
bdNamErr	Bad file name
dupFNErr	Duplicate file name
dirFulErr	Directory full
extFSErr	External file system
ioErr	Disk I/O error
nsvErr	No such volume
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

PBCreate creates a new file having the name ioNamePtr and the version number ioVersNum, on the specified volume. The new file is unlocked and empty. Its modification and creation dates are set to the time of the system clock. The application should call PBSetFInfo to fill in the information needed by the Finder.

```
FUNCTION PBOpen (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _Open

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
<--	24	ioRefNum	word
-->	26	ioVersNum	byte
-->	27	ioPermssn	byte
-->	28	ioMisc	pointer

Result codes

noErr	No error
bdNamErr	Bad file name
extFSErr	External file system
fnfErr	File not found
ioErr	Disk I/O error
mFulErr	Memory full
nsvErr	No such volume
opWrErr	File already open for writing
tmfoErr	Too many files open

PBOpen creates an access path to the file having the name ioNamePtr and the version number ioVersNum, on the specified volume. A path reference number is returned in ioRefNum.

IOMisc either points to a 522-byte portion of memory to be used as the access path's buffer, or is NIL if you want the volume buffer to be used instead.

(warning)

All access paths to a single file that's opened multiple times should share the same buffer so that they will read and write the same data.

IOPermssn specifies the path's read/write permission. A path can be opened for writing even if it accesses a file on a locked volume, and an error won't be returned until a PBWrite, PBSetEOF, or PBAllocate call is made.

If you attempt to open a locked file for writing, PBOpen will return opWrErr as its function result. If you attempt to open a file for writing and it already has an access path that allows writing, PBOpen will return the reference number of the existing access path in ioRefNum and opWrErr as its function result.

```
FUNCTION PBOpenRF (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _OpenRF

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
<--	24	ioRefNum	word
-->	26	ioVersNum	byte
-->	27	ioPermssn	byte
-->	28	ioMisc	pointer

Result codes

noErr	No error
bdNamErr	Bad file name
extFSErr	External file system
fnfErr	File not found
ioErr	Disk I/O error
mFulErr	Memory full
nsvErr	No such volume
opWrErr	File already open for writing
permErr	Open permission doesn't allow reading
tmfoErr	Too many files open

PBOpenRF is identical to PBOpen, except that it opens the file's resource fork instead of its data fork.

```
FUNCTION PBRead (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _Read

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word
-->	32	ioBuffer	pointer
-->	36	ioReqCount	long word
<--	40	ioActCount	long word
-->	44	ioPosMode	word
<->	46	ioPosOffset	long word

Result codes

noErr	No error
eofErr	End-of-file
extFSErr	External file system
fnOpnErr	File not open
ioErr	Disk I/O error
paramErr	Negative ioReqCount
rfNumErr	Bad reference number

PBRead attempts to read ioReqCount bytes from the open file whose access path is specified by ioRefNum, and transfer them to the data buffer pointed to by ioBuffer. If you try to read past the logical end-of-file, PBRead moves the mark to the end-of-file and returns eofErr as its function result. After the read operation is completed, the mark is returned in ioPosOffset and the number of bytes actually read is returned in ioActCount.

(note)

Advanced programmers: IOPosMode contains the newline character (if any), and indicates whether the read should begin relative to the beginning of the file, the mark, or the end-of-file. The byte offset from the position indicated by ioPosMode, where the read should actually begin, is given by ioPosOffset. If a newline character is not specified, the data will be read one byte at a time until ioReqCount bytes have been read or the end-of-file is reached. If a newline character is specified, the data will be read one byte at a time until the newline character is encountered, the end-of-file is reached, or ioReqCount bytes have been read.

```
FUNCTION PBWrite (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _Write

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word
-->	32	ioBuffer	pointer
-->	36	ioReqCount	long word
<--	40	ioActCount	long word
-->	44	ioPosMode	word
-->	46	ioPosOffset	long word

Result codes

noErr	No error
dskFulErr	Disk full
fLckdErr	File locked
fnOpnErr	File not open
ioErr	Disk I/O error
paramErr	Negative ioReqCount
posErr	Position is beyond end-of-file
rfNumErr	Bad reference number
vLckdErr	Software volume lock
wPrErr	Hardware volume lock
wrPermErr	Read/write or open permission doesn't allow writing

PBWrite takes ioReqCount bytes from the buffer pointed to by ioBuffer and attempts to write them to the open file whose access path is specified by ioRefNum. After the write operation is completed, the mark is returned in ioPosOffset, and the number of bytes actually written is returned in ioActCount.

IOPosMode indicates whether the write should begin relative to the beginning of the file, the mark, or the end-of-file. The byte offset from the position indicated by ioPosMode, where the write should actually begin, is given by ioPosOffset.

```
FUNCTION PBGetFPos (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _GetFPos

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	22	ioRefNum	word
<--	36	ioReqCount	long word
<--	40	ioActCount	long word
<--	44	ioPosMode	word
<--	46	ioPosOffset	long word

Result codes

noErr	No error
extFSErr	External file system
fnOpnErr	File not open
ioErr	Disk I/O error
rfNumErr	Bad reference number

PBGetFPos returns, in ioPosOffset, the mark of the open file whose access path is specified by ioRefNum. PBGetFPos sets ioReqCount, ioActCount, and ioPosMode to 0.

```
FUNCTION PBSetFPos (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _SetFPos

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	22	ioRefNum	word
-->	44	ioPosMode	word
-->	46	ioPosOffset	long word

Result codes

noErr	No error
eofErr	End-of-file
extFSErr	External file system
fnOpnErr	File not open
ioErr	Disk I/O error
posErr	Tried to position before start of file
rfNumErr	Bad reference number

PBSetFPos sets the mark of the open file whose access path is specified by ioRefNum, to the position specified by ioPosMode and ioPosOffset. IoPosMode indicates whether the mark should be set relative to the beginning of the file, the mark, or the logical end-of-file. The byte offset from the position given by ioPosMode, where the mark should actually be set, is given by ioPosOffset. If you try to set the mark past the logical end-of-file, PBSetFPos moves the mark to the end-of-file and returns eofErr as its function result.

```
FUNCTION PBGetEOF (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _GetEOF

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	22	ioRefNum	word
<--	28	ioMisc	long word

Result codes

noErr	No error
extFSErr	External file system
fnOpnErr	File not open
ioErr	Disk I/O error
rfNumErr	Bad reference number

PBGetEOF returns, in ioMisc, the logical end-of-file of the open file whose access path is specified by ioRefNum.

```
FUNCTION PBSetEOF (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _SetEOF

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	22	ioRefNum	word
-->	28	ioMisc	long word

Result codes

noErr	No error
dskFulErr	Disk full
extFSErr	External file system
fLckdErr	File locked
fnOpnErr	File not open
ioErr	Disk I/O error
rfNumErr	Bad reference number
vLckdErr	Software volume lock
wPrErr	Hardware volume lock
wrPermErr	Read/write or open permission doesn't allow writing

PBSetEOF sets the logical end-of-file of the open file whose access path is specified by ioRefNum, to ioMisc. If the logical end-of-file is set beyond the physical end-of-file, the physical end-of-file is set to one byte beyond the end of the next free allocation block; if there isn't enough space on the volume, no change is made, and PBSetEOF returns dskFulErr as its function result. If ioMisc is Ø, all space on the volume occupied by the file is released.

```
FUNCTION PBAllocate (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _Allocate

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	22	ioRefNum	word
-->	36	ioReqCount	long word
<--	40	ioActCount	long word

Result codes

noErr	No error
dskFulErr	Disk full
fLckdErr	File locked
fnOpnErr	File not open
ioErr	Disk I/O error
rfNumErr	Bad reference number
vLckdErr	Software volume lock
wPrErr	Hardware volume lock
wrPermErr	Read/write or open permission doesn't allow writing

PBAllocate adds ioReqCount bytes to the open file whose access path is specified by ioRefNum, and sets the physical end-of-file to one byte beyond the last block allocated. The number of bytes allocated is always rounded up to the nearest multiple of the allocation block size, and returned in ioActCount. If there isn't enough empty space on the volume to satisfy the allocation request, PBAllocate allocates the rest of the space on the volume and returns dskFulErr as its function result.

```
FUNCTION PBFlshFile (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _FlushFile

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	22	ioRefNum	word

Result codes

noErr	No error
extFSErr	External file system
fnfErr	File not found
fnOpnErr	File not open
ioErr	Disk I/O error
nsvErr	No such volume
rfNumErr	Bad reference number

PBFlshFile writes the contents of the access path buffer indicated by ioRefNum to the volume, and updates the file's entry in the file directory.

(warning)
Some information stored on the volume won't be correct until PBFlshVol is called.

```
FUNCTION PBClose (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _Close

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word

Result codes

noErr	No error
extFSErr	External file system
fnfErr	File not found
fnOpnErr	File not open
ioErr	Disk I/O error
nsvErr	No such volume
rfNumErr	Bad reference number

PBClose writes the contents of the access path buffer specified by ioRefNum to the volume and removes the access path.

(warning)
Some information stored on the volume won't be correct until PBFlshVol is called.

Changing Information About Files

All of the routines described in this section affect both forks of a file.

```
FUNCTION PBGetFInfo (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _GetFileInfo

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
<--	24	ioRefNum	word
-->	26	ioVersNum	byte
-->	28	ioFDirIndex	word
<--	30	ioFlAttrib	byte
<--	31	ioFlVersNum	byte
<--	32	ioFndrInfo	16 bytes
<--	48	ioFlNum	long word
<--	52	ioFlStBlk	word
<--	54	ioFlLgLen	long word
<--	58	ioFlPyLen	long word
<--	62	ioFlRStBlk	word
<--	64	ioFlRLgLen	long word
<--	68	ioFlRPyLen	long word
<--	72	ioFlCrDat	long word
<--	76	ioFlMdDat	long word

Result codes

noErr	No error
bdNamErr	Bad file name
extFSErr	External file system
fnfErr	File not found
ioErr	Disk I/O error
nsvErr	No such volume
paramErr	No default volume

PBGetFInfo returns information about the specified file. If ioFDirIndex is positive, the File Manager returns information about the file whose file number is ioFDirIndex on the specified volume (see the section "Data Organization on Volumes" if you're interested in using this method). If ioFDirIndex is negative or 0, the File Manager returns information about the file having the name ioNamePtr and the version number ioVersNum, on the specified volume. Unless ioNamePtr is NIL, ioNamePtr returns a pointer to the name of the file. If the file is open, the reference number of the first access path found is returned in ioRefNum.

```
FUNCTION PBSetFInfo (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _SetFileInfo

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
-->	26	ioVersNum	byte
-->	32	ioFndrInfo	16 bytes
-->	72	ioFlCrDat	long word
-->	76	ioFlMdDat	long word

Result codes

noErr	No error
bdNamErr	Bad file name
extFSErr	External file system
fLckdErr	File locked
fnfErr	File not found
ioErr	Disk I/O error
nsvErr	No such volume
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

PBSetFInfo sets information (including creation and modification dates, and information needed by the Finder) about the file having the name ioNamePtr and the version number ioVersNum on the specified volume. You should call PBGetFInfo just before PBSetFInfo, so the current information is present in the parameter block.

```
FUNCTION PBSetFLock (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _SetFilLock

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
-->	26	ioVersNum	byte

Result codes

noErr	No error
extFSErr	External file system
fnfErr	File not found
ioErr	Disk I/O error
nsvErr	No such volume
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

PBSetFLock locks the file having the name ioNamePtr and the version number ioVersNum on the specified volume. Access paths currently in use aren't affected.

```
FUNCTION PBRstFLock (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _RstFilLock

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
-->	26	ioVersNum	byte

Result codes

noErr	No error
extFSErr	External file system
fnfErr	File not found
ioErr	Disk I/O error
nsvErr	No such volume
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

PBRstFLock unlocks the file having the name ioNamePtr and the version number ioVersNum on the specified volume. Access paths currently in use aren't affected.

```
FUNCTION PBSetFVers (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _SetFilType

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
-->	26	ioVersNum	byte
-->	28	ioMisc	byte

Result codes

noErr	No error
bdNamErr	Bad file name
dupFNErr	Duplicate file name and version
extFSErr	External file system
fLckdErr	File locked
fnfErr	File not found
nsvErr	No such volume
ioErr	Disk I/O error
paramErr	No default volume
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

PBSetFVers changes the version number of the file having the name ioNamePtr and version number ioVersNum on the specified volume, to ioMisc. Access paths currently in use aren't affected.

(warning)
The Resource Manager and Segment Loader operate only on files with version number Ø; changing the version number of a file to a nonzero number will prevent them from operating on it.

```
FUNCTION PBRename (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _Rename

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
-->	26	ioVersNum	byte
-->	28	ioMisc	pointer

Result codes

noErr	No error
bdNamErr	Bad file name
dirFulErr	Directory full
dupFNErr	Duplicate file name and version
extFSErr	External file system
fLckdErr	File locked
fnfErr	File not found
fsRnErr	Renaming difficulty
ioErr	Disk I/O error
nsvErr	No such volume
paramErr	No default volume
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

Given a file name in ioNamePtr and a version number in ioVersNum, Rename changes the name of the specified file to ioMisc; given a volume name in ioNamePtr or a volume reference number in ioVRefNum, it changes the name of the specified volume to ioMisc. Access paths currently in use aren't affected.

```
FUNCTION PBDelete (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _Delete

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
-->	22	ioVRefNum	word
-->	26	ioVersNum	byte

Result codes

noErr	No error
bdNamErr	Bad file name
extFSErr	External file system
fBsyErr	File busy
fLckdErr	File locked
fnfErr	File not found
nsvErr	No such volume
ioErr	Disk I/O error
vLckdErr	Software volume lock
wPrErr	Hardware volume lock

PBDelete removes the closed file having the name ioNamePtr and the version number ioVersNum, from the specified volume.

(note)
This function will delete **both** forks of the file.

DATA ORGANIZATION ON VOLUMES

This section explains how information is organized on volumes. Most of the information is accessible only through assembly language, but some advanced Pascal programmers may be interested.

The File Manager communicates with device drivers that read and write data via block-level requests to devices containing Macintosh-initialized volumes. (Macintosh-initialized volumes are volumes initialized by the Disk Initialization Package.) The actual type of volume and device is unimportant to the File Manager; the only requirements are that the volume was initialized by the Disk Initialization Package and that the device driver is able to communicate via block-level requests.

The 3 1/2-inch built-in and optional external drives are accessed via the Disk Driver. If you want to use the File Manager to access files on Macintosh-initialized volumes on other types of devices, you must write a device driver that can read and write data via block-level requests to the device on which the volume will be mounted. If you want to access files on nonMacintosh-initialized volumes, you must write your own external file system (see the section "Using an External File System").

The information on all block-formatted volumes is organized in logical blocks and allocation blocks. Logical blocks contain a number of bytes of standard information (512 bytes on Macintosh-initialized volumes), and an additional number of bytes of information specific to the disk driver (12 bytes on Macintosh-initialized volumes). Allocation blocks are composed of any integral number of logical blocks, and are simply a means of grouping logical blocks together in more convenient parcels.

The remainder of this section applies only to Macintosh-initialized volumes. NonMacintosh-initialized volumes must be accessed via an external file system, and the information on them must be organized by an external initializing program.

A Macintosh-initialized volume contains information needed to start up the system in logical blocks Ø and 1 (Figure 6). Logical block 2 of the volume begins the master directory block. The master directory block contains volume information and the volume allocation block map, which records whether each block on the volume is unused or what part of a file it contains data from.

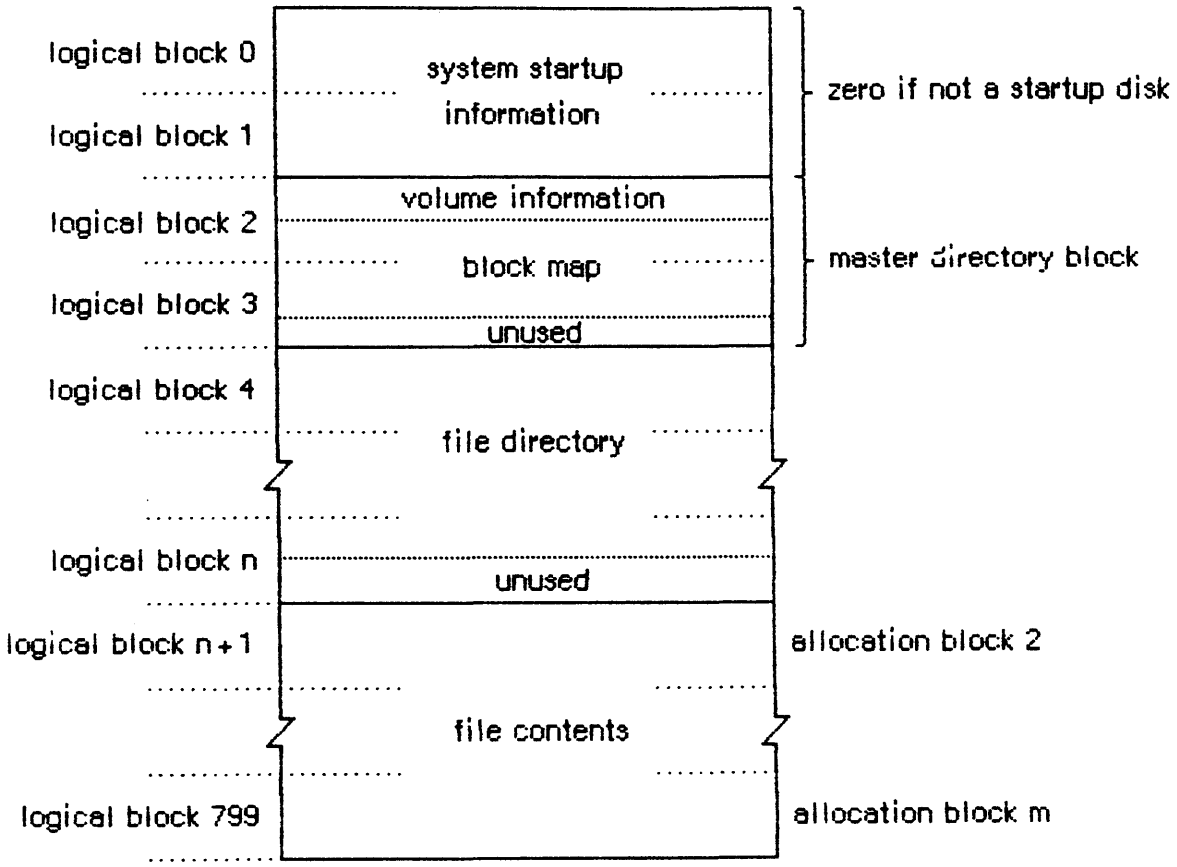


Figure 6. A 4ØØK-Byte Volume With 1K-Byte Allocation Blocks

The master directory "block" always occupies two blocks--the Disk Initialization Package varies the allocation block size as necessary to achieve this constraint.

In the next logical block following the block map begins the file directory, which contains descriptions and locations of all the files on the volume. The rest of the logical blocks on the volume contain files or garbage (such as parts of deleted files). The exact format of the volume information, volume allocation block map, file directory, and files is explained in the following sections.

Volume Information

The volume information is contained in the first 64 bytes of the master directory block (Figure 7). This information is written on the volume when it's initialized, and modified thereafter by the File Manager.

byte 0	drSigWord (word)	always $D2D7
2	drCrDate (long word)	date and time of intialization
6	drLsBkUp (long word)	date and time of last backup
10	drAtrb (word)	volume attributes
12	drNmFls (word)	number of files in file directory
14	drDirSt (word)	first logical block of file directory
16	drBlLen (word)	number of logical blocks in file directory
18	drNmAlBlks (word)	number of allocation blocks on volume
20	drAlBlkSiz (long word)	size of allocation blocks
24	drClpSiz (long word)	number of bytes to allocate
28	drAlBlSt (word)	logical block number of first allocation block
30	drNxtFNum (long word)	next unused file number
34	drFreeBks (word)	number of unused allocation blocks
36	drVN (byte)	length of volume name
	drVN+1 (bytes)	characters of volume name

Figure 7. Volume Information

DrAtrb contains the volume attributes. Its bits, if set, indicate the following:

Bit	Meaning
7	Volume is locked by hardware
15	Volume is locked by software

DrClpSiz contains the minimum number of bytes to allocate each time the Allocate function is called, to minimize fragmentation of files; it's always a multiple of the allocation block size. DrNxtFNum contains the next unused file number (see the "File Directory" section below for an explanation of file numbers).

Volume Allocation Block Map

The volume allocation block map represents every allocation block on the volume with a 12-bit entry indicating whether the block is unused or allocated to a file. It begins in the master directory block at the byte following the volume information, and continues for as many logical blocks as needed. For example, a 4ØØK-byte volume with a 1Ø-block file directory and 1K-byte allocation blocks would have a 591-byte block map.

The first entry in the block map is for block number 2; the block map doesn't contain entries for the startup blocks. Each entry specifies whether the block is unused, whether it's the last block in the file, or which allocation block is next in the file:

Entry	Meaning
Ø	Block is unused
1	Block is the last block of the file
2..4Ø95	Number of next block in the file

For instance, assume that there's one file on the volume, stored in allocation blocks 8, 11, 12, and 17; the first 16 entries of the block map would read

Ø Ø Ø Ø Ø Ø 11 Ø Ø 12 17 Ø Ø Ø Ø 1

The first allocation block on a volume typically follows the file directory. The first allocation block is number 2 because of the special meaning of numbers Ø and 1.

(note)

As explained below, it's possible to begin the allocation blocks immediately following the master directory block and place the file directory somewhere within the allocation blocks. In this case, the allocation blocks occupied by the file directory must be marked with $FFF's in the allocation block map.

File Directory

The file directory contains an entry for each file. Each entry lists information about one file on the volume, including its name and location. Each file is listed by its own unique file number, which the File Manager uses to distinguish it from other files on the volume.

A file directory entry contains 51 bytes plus one byte for each character in the file name (Figure 8); if the file names average 2Ø characters, a directory can hold seven file entries per logical block. Entries are always an integral number of words and don't cross logical block boundaries. The length of a file directory depends on the maximum number of files the volume can contain; for example, on a 4ØØK-byte volume the file directory occupies 12 logical blocks.

The file directory conventionally follows the block map and precedes the allocation blocks, but a volume-initializing program could actually place the file directory anywhere within the allocation blocks as long as the blocks occupied by the file directory are marked with $FFF's in the block map.

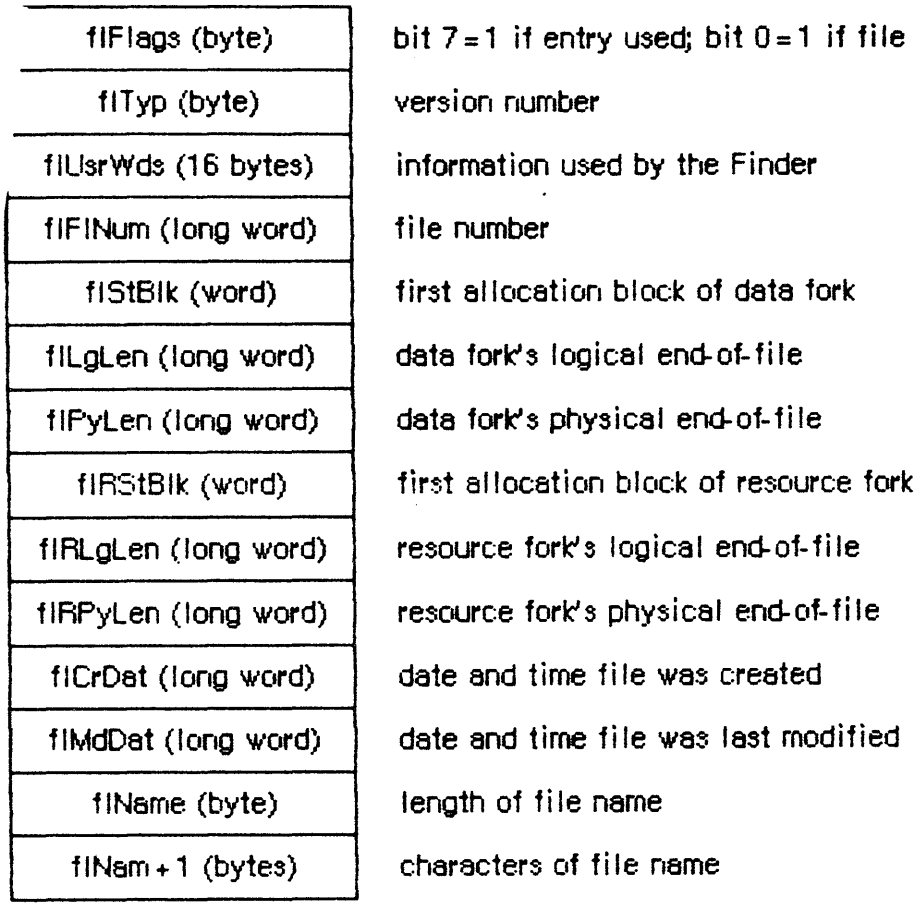


Figure 8. A File Directory Entry

FlStBlk and flRStBlk are Ø if the data or resource fork doesn't exist. FlCrDat and flMdDat are given in seconds since 12:ØØ AM, January 1, 19Ø4.

Each time a new file is created, an entry for the new file is placed in the file directory. Each time a file is deleted, its entry in the file directory is cleared, and all blocks used by that file on the volume are released.

File Tags on Volumes

As mentioned previously, logical blocks contain 512 bytes of standard information preceded by 12 bytes of file tags (Figure 9). The file tags are designed to allow easy reconstruction of files from a volume whose directory or other file-access information has been destroyed.

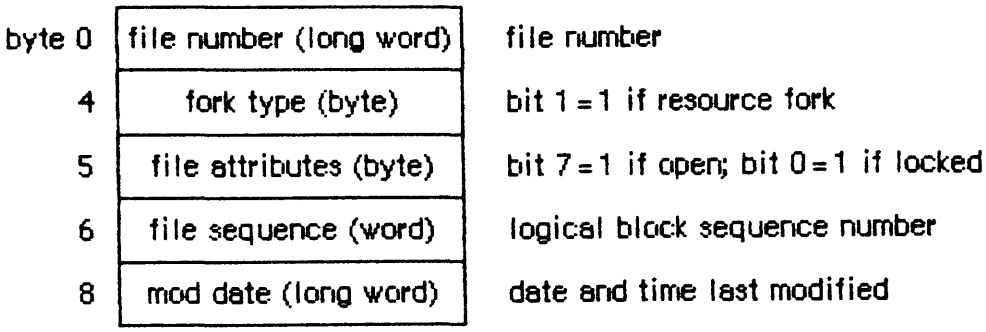


Figure 9. File Tags on Volumes

The file sequence indicates which relative portion of a file the block contains--the first logical block of a file has a sequence number of Ø, the second a sequence number of 1, and so on.

DATA STRUCTURES IN MEMORY

This section describes the memory data structures used by the File Manager and any external file system that accesses files on Macintosh-initialized volumes. Most of this data is accessible only through assembly language, but some advanced Pascal programmers may be interested.

The data structures in memory used by the File Manager and all external file systems include:

- the file I/O queue, listing the currently executing routine (if any), and any asynchronous routines awaiting execution
- the volume-control-block queue, listing information about each mounted volume
- copies of volume allocation block maps; one for each on-line volume
- the file-control-block buffer, listing information about each access path
- volume buffers; one for each on-line volume
- optional access path buffers; one for each access path
- the drive queue, listing information about each drive connected to the Macintosh

The File I/O Queue

The file I/O queue is a standard Operating System queue (described in the appendix) that contains a list of all asynchronous routines awaiting execution. Each time a routine is called, an entry is placed in the queue; each time a routine is completed, its entry is removed from the queue.

The file I/O queue uses entries of type ioQType, each of which consists of a parameter block for the routine that was called. The structure of this block is shown in part below:

```
TYPE ParamBlockRec = RECORD
                       qLink:     QElemPtr; {next queue entry}
                       qType:     INTEGER;  {queue type}
                       ioTrap:    INTEGER;  {routine trap}
                       ioCmdAddr: Ptr;      {routine address}
                         . . .              {rest of block}
                     END;
```

QLink points to the next entry in the queue, and qType indicates the queue type, which must always be ORD(ioQType). IOTrap and ioCmdAddr contain the trap word and address of the File Manager routine that was called. You can get a pointer to the file I/O queue by calling the File Manager function GetFSQHdr.

```
FUNCTION GetFSQHdr : QHdrPtr;  [Pascal only]
```

GetFSQHdr returns a pointer to the file I/O queue.

Assembly-language note: To access the contents of the file I/O queue from assembly language, you can use offsets from the address of the global variable fsQHdr. Bit 7 of the queue flags is set if there are any entries in the queue; you can use the global constant qInUse to test the value of bit 7.

Volume Control Blocks

Each time a volume is mounted, its volume information is read from the volume and used to build a new volume control block in the volume-control-block queue (unless an ejected or off-line volume is being remounted). A copy of the volume block map is also read from the volume and placed in the system heap, and a volume buffer is created on the system heap.

The volume-control-block queue is a list of the volume control blocks for all mounted volumes, maintained on the system heap. It's a standard Operating System queue (described in the appendix), and each entry in the volume-control-block queue is a volume control block. A volume control block is a 94-byte nonrelocatable block that contains volume-specific information, including the first 64 bytes of the master directory block (bytes 8 to 72 of the volume control block match bytes Ø to 64 of the volume information). It has the following structure:

```
TYPE VCB = RECORD
             qLink:       QElemPtr;    {next queue entry}
             qType:       INTEGER;     {not used}
             vcbFlags:    INTEGER;     {bit 15=1 if dirty}
             vcbSigWord:  INTEGER;     {always $D2D7}
             vcbCrDate:   LongInt;     {date volume was initialized}
             vcbLsBkUp:   LongInt;     {date of last backup}
             vcbAtrb:     INTEGER;     {volume attributes}
             vcbNmFls:    INTEGER;     {number of files in directory}
             vcbDirSt:    INTEGER;     {directory's first block}
             vcbBlLn:     INTEGER;     {length of file directory}
             vcbNmBlks:   INTEGER;     {number of allocation blocks}
             vcbAlBlkSiz: LongInt;     {size of allocation blocks}
             vcbClpSiz:   LongInt;     {number of bytes to allocate}
             vcbAlBlSt:   INTEGER;     {first block in block map}
             vcbNxtFNum:  LongInt;     {next unused file number}
             vcbFreeBks:  INTEGER;     {number of unused blocks}
             vcbVN:       STRING[27]; {volume name}
             vcbDrvNum:   INTEGER;     {drive number}
             vcbDRefNum:  INTEGER;     {driver reference number}
             vcbFSID:     INTEGER;     {file system identifier}
             vcbVRefNum:  INTEGER;     {volume reference number}
             vcbMAdr:     Ptr;         {location of block map}
             vcbBufAdr:   Ptr;         {location of volume buffer}
             vcbMLen:     INTEGER;     {number of bytes in block map}
             vcbDirIndex: INTEGER;     {used internally}
             vcbDirBlk:   INTEGER      {used internally}
           END;
```

Bit 15 of vcbFlags is set if the volume information has been changed by a routine call since the volume was last affected by a FlushVol call. VCBAtr contains the volume attributes. Each bit, if set, indicates the following:

Bit	Meaning
Ø-2	Inconsistencies were found between the volume information and the file directory when the volume was mounted
6	Volume is busy (one or more files are open)
7	Volume is locked by hardware
15	Volume is locked by software

VCBDirSt contains the number of the first logical block of the file directory; vcbNmBlks, the number of allocation blocks on the volume; vcbAlBlSt, the number of the first logical block in the block map; and vcbFreeBks, the number of unused allocation blocks on the volume.

VCBDrvNum contains the drive number of the drive on which the volume is mounted; vcbDRefNum contains the driver reference number of the driver used to access on volume is mounted. When a mounted volume is placed off-line, vcbDrvNum is cleared. When ejected, vcbDrvNum is cleared and vcbDRefNum is set to the negative of vcbDrvNum (becoming a positive number). VCBFSID identifies the file system handling the volume; it's Ø for volumes handled by the File Manager, and nonzero for volumes handled by other file systems.

When a volume is placed off-line, its buffer and block map are deallocated. When a volume is unmounted, its volume control block is removed from the volume-control-block queue.

You can get a pointer to the volume-control-block queue by calling the File Manager function GetVCBQHdr.

```
FUNCTION GetVCBQHdr : QHdrPtr;  [Pascal only]
```

GetVCBQHdr returns a pointer to the volume-control-block queue.

Assembly-language note: To access the contents of the volume-control-block queue from assembly language, you can use offsets from the address of the global variable vcbQHdr. Bit 7 of the queue flags is set if there are any entries in the queue; you can use the global constant qInUse to test the value of bit 7. The default volume's volume control block is pointed to by the global variable defVCBPtr.

File Control Blocks

Each time a file is opened, the file's directory entry is used to build a 3Ø-byte file control block in the file-control-block buffer, which contains information about all access paths. The file-control-block buffer can contain up to 12 file control blocks (since up to 12 paths can be open at once), and is a 362-byte (2 + 3Ø bytes*12 paths) nonrelocatable block on the system heap (see Figure 1Ø).

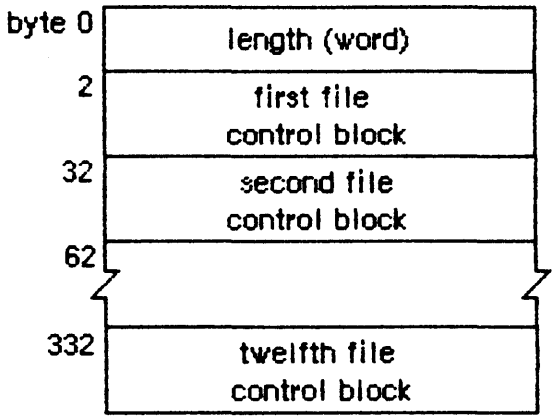


Figure 1Ø. The File-Control-Block Buffer

You can refer to the file-control-block buffer by using the global variable fcbSPtr, which points to the length word. Each file control block contains 3Ø bytes of information about an access path (Figure 11).

byte 0	fcbFlNum (long word)	file number
4	fcbMdFlByt (byte)	flags
5	fcbTypByt (byte)	version number
6	fcbSBlk (word)	first allocation block of file
8	fcbEOF (long word)	logical end-of-file
12	fcbPLen (long word)	physical end-of-file
16	fcbCrPs (long word)	mark
20	fcbVPtr (pointer)	location of volume control block
24	fcbBfAdr (pointer)	location of access path buffer
28	fcbFlPos (word)	for internal use of File Manager

Figure 11. A File Control Block

Bit 7 of fcbMdRByt is set if the file has been changed since it was last flushed; bit 1 is set if the entry describes a resource fork; bit Ø is set if data can be written to the file.

Files Tags in Memory

As mentioned previously, logical blocks on Macintosh-initialized volumes contain 12 bytes of file tags. Normally, you'll never need to know about file tags, and the File Manager will let you read and write only the 512 bytes of standard information in each logical block. The File Manager automatically removes the file tags from each logical block it reads into memory (Figure 12) and places them at the location referred to by the global variable tagData + 2. It replaces the last four bytes of the file tags with the number of the logical block from which the file was read (leaving a total of ten bytes).

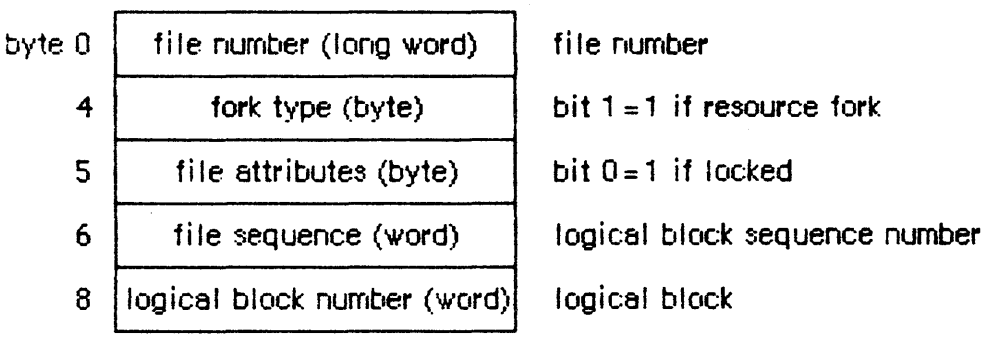


Figure 12. File Tags in Memory

(note)

Access path buffers and volume buffers are 522 bytes long in order to contain the ten bytes of file tags and 512 bytes of standard information.

The Drive Queue

Disk drives connected to the Macintosh are opened when the system starts up, and information describing each is placed in the drive queue. It's a standard Operating System queue (described in the appendix), and each entry in the drive queue has the following structure:

```
TYPE DrvQEl = RECORD
                { flags:     LongInt; }
                  qLink:     QElemPtr; {next queue entry}
                  qType:     INTEGER;  {not used}
                  dQDrive:   INTEGER;  {drive number}
                  dQRefNum:  INTEGER;  {driver reference number}
                  dQFSID:    INTEGER;  {file-system identifier}
                  dQDrvSize: INTEGER   {optional:  number of blocks}
                END;
```

QDrvNum contains the drive number of the drive on which the volume is mounted; qDRefNum contains the driver reference number of the driver

controlling the device on which the volume is mounted. QFSID identifies the file system handling the volume in the drive; it's Ø for volumes handled by the File Manager, and nonzero for volumes handled by other file systems. If the volume isn't a 3-1/2 inch disk, dQDrvSize contains the number of 512-byte blocks on the volume mounted in this drive; if the volume is a 3-1/2 inch disk, this field isn't used.

Assembly-language note: The first four bytes in a drive queue entry are accessible only from assembly language, and contain the following:

Byte	Contents
Ø	Bit 7=1 if volume is locked
1	Ø if no disk in drive; 1 or 2 if disk in drive; 8 if nonejectable disk in drive; $FC-$FF if disk was ejected within last 1.5 seconds
2	used internally during system startup
3	Bit 7=Ø if disk is single-sided

You can get a pointer to the drive queue by calling the File Manager function GetDrvQHdr:

```
FUNCTION GetDrvQHdr : QHdrPtr;  [Pascal only]
```

GetDrvQHdr returns a pointer to the qFlags field.

Assembly-language note: To access the contents of the drive queue from assembly language, you can use offsets from the address of the global variable drvQHdr.

The drive queue can support any number of drives, limited only by memory space.

USING AN EXTERNAL FILE SYSTEM

The File Manager is used to access files on Macintosh-initialized volumes. If you want to access files on nonMacintosh-initialized volumes, you must write your own external file system and volume-initializing program. After the external file system has been written, it must be used in conjunction with the File Manager as described in this section.

Before any File Manager routines are called, you must place the memory location of the external file system in the global variable toExtFS; and link the drive(s) accessed by your file system into the drive queue. As each nonMacintosh-initialized volume is mounted, you must create your own volume control block for each mounted volume and link each one into the volume-control-block queue. As each access path is opened, you must create your own file control block and add it to the file-control-block buffer.

All SetVol, GetVol, and GetVolInfo calls then can be handled by the File Manager via the volume-control-block queue and drive queue; external file systems needn't support these calls.

When an application calls any other File Manager routine accessing a nonMacintosh-initialized volume, the File Manager passes control to the address contained in toExtFS (if toExtFS is Ø, the File Manager returns directly to the application with the result code extFSErr). The external file system must then use the information in the file I/O queue to handle the call as it wishes, set the result code noErr, and return control to the File Manager. Control is passed to an external file system for the following specific routine calls:

- for MountVol if the drive queue entry for the requested drive has a nonzero file-system identifier

- for Create, Open, OpenRF, GetFileInfo, SetFileInfo, SetFilLock, RstFilLock, SetFilType, Rename, Delete, FlushVol, Eject, OffLine, and UnmountVol, if the volume control block for the requested file or volume has a nonzero file-system identifier

- for Close, Read, Write, Allocate, GetEOF, SetEOF, GetFPos, SetFPos, and FlushFile, if the file control block for the requested file points to a volume control block with a nonzero file-system identifier

APPENDIX -- OPERATING SYSTEM QUEUES

*** This appendix will eventually be part of the Operating System Utilities manual. ***

Some of the information used by the Operating System is stored in data structures called queues. A queue is a list of identically structured entries linked together by pointers. Queues are used to keep track of vertical retrace tasks, I/O requests, disk drives, events, and mounted volumes.

The structure of a standard Operating System queue is as follows:

```
TYPE QHdr = RECORD
              qFlags: INTEGER;  {queue flags}
              qHead:  QElemPtr; {first queue entry}
              qTail:  QElemPtr  {last queue entry}
            END;

     QHdrPtr = ^QHdr;
```

QFlags contains information that's different for each queue type. QHead points to the first entry in the queue, and qTail points to the last entry in the queue. The entries within each type of queue are different, since each type of queue contains different information. The Operating System uses the following variant record to access queue entries:

```
TYPE QTypes = (dummyType,
               vType,     {vertical retrace queue}
               ioQType,   {I/O request queue}
               drvQType,  {drive queue}
               evType,    {event queue}
               fsQType);  {volume-control-block queue}

     QElem = RECORD
               CASE QTypes OF
                 (vblQElem: VBLTask);
                 (ioQElem:  ParamBlockRec);
                 (drvQElem: DrvQEl);
                 (evQElem:  EvQEl);
                 (vcbQElem: VCB)
             END;

     QElemPtr = ^QElem;
```

The exact structure of the entries in each type of Operating System queue is described in the manual that discusses that queue in detail.

Assembly-language note: The values given in the Pascal QTypes set are available to assembly-language programmers as the global

constants vType, ioQType, evType, and fsQType (there is no global constant corresponding to drvQType).

SUMMARY OF THE FILE MANAGER

Constants

```
CONST { Flags in file information used by the Finder }

      fHasBundle = 32; {set if file has a bundle}
      fInvisible = 64; {set if file's icon is invisible}
      fTrash     = -3; {file is in trash window}
      fDesktop   = -2; {file is on desktop}
      fDisk      =  Ø; {file is in disk window}

      { Values for posMode and ioPosMode }

      fsAtMark    = Ø; {at current position of mark }
                       { (posOff or ioPosOffset ignored)}
      fsFromStart = 1; {offset relative to beginning of file}
      fsFromLEOF  = 2; {offset relative to logical end-of-file}
      fsFromMark  = 3; {offset relative to current mark}

      { Values for requesting read/write access }

      fsCurPerm  = Ø; {whatever is currently allowed}
      fsRdPerm   = 1; {request to read only}
      fsWrPerm   = 2; {request to write only}
      fsRdWrPerm = 3; {request to read and write}
```

(See also the result codes at end of this summary.)

Data Structures

```
TYPE FInfo = RECORD
               fdType:     OSType;  {file type}
               fdCreator:  OSType;  {file's creator}
               fdFlags:    INTEGER; {flags}
               fdLocation: Point;   {file's location}
               fdFldr:     INTEGER  {file's window}
             END;

     ParamBlkPtr    = ^ParamBlockRec;

     ParamBlkType  = (ioParam, fileParam, volumeParam, cntrlParam);
```

```
ParamBlockRec = RECORD
 qLink:         QElemPtr;  {next queue entry}
 qType:         INTEGER;   {queue type}
 ioTrap:        INTEGER;   {routine trap}
 ioCmdAddr:     Ptr;       {routine address}
 ioCompletion:  ProcPtr;   {completion routine}
 ioResult:      OSErr;     {result code}
 ioNamePtr:     StringPtr; {volume or file name}
 ioVRefNum:     INTEGER;   {volume reference or }
                           { drive number}
 CASE ParamBlkType OF
  ioParam:
   (ioRefNum:     INTEGER;    {path reference number}
    ioVersNum:    SignedByte; {version number}
    ioPermssn:    SignedByte; {read/write permission}
    ioMisc:       Ptr;        {miscellaneous}
    ioBuffer:     Ptr;        {data buffer}
    ioReqCount:   LongInt;    {requested number of bytes}
    ioActCount:   LongInt;    {actual number of bytes}
    ioPosMode:    INTEGER;    {newline character and type of }
                              { positioning operation}
    ioPosOffset:  LongInt);   {size of positioning offset}
  fileParam:
   (ioFRefNum:    INTEGER;    {path reference number}
    ioFVersNum:   SignedByte; {version number}
    filler1:      SignedByte; {not used}
    ioFDirIndex:  INTEGER;    {file number}
    ioFlAttrib:   SignedByte; {file attributes}
    ioFlVersNum:  SignedByte; {version number}
    ioFlFndrInfo: FInfo;      {information used by the Finder}
    ioFlNum:      LongInt;    {file number}
    ioFlStBlk:    INTEGER;    {first allocation block of data fork}
    ioFlLgLen:    LongInt;    {logical end-of-file of data fork}
    ioFlPyLen:    LongInt;    {physical end-of-file of data fork}
    ioFlRStBlk:   INTEGER;    {first allocation block of resource fork}
    ioFlRLgLen:   LongInt;    {logical end-of-file of resource fork}
    ioFlRPyLen:   LongInt;    {physical end-of-file of resource fork}
    ioFlCrDat:    LongInt;    {date and time of creation}
    ioFlMdDat:    LongInt);   {date and time of last modification}
  volumeParam:
   (filler2:      LongInt;  {not used}
    ioVolIndex:   INTEGER;  {volume index}
    ioVCrDate:    LongInt;  {date and time of initialization}
    ioVLsBkUp:    LongInt;  {date and time of last volume backup}
    ioVAtrb:      INTEGER;  {bit 15=1 if volume locked}
    ioVNmFls:     INTEGER;  {number of files in file directory}
    ioVDirSt:     INTEGER;  {first block of file directory}
    ioVBlLn:      INTEGER;  {number of blocks in file directory}
    ioVNmAlBlks:  INTEGER;  {number of allocation blocks on volume}
    ioVAlBlkSiz:  LongInt;  {number of bytes per allocation block}
    ioVClpSiz:    LongInt;  {number of bytes to allocate}
    ioAlBlSt:     INTEGER;  {first block in volume block map}
    ioVNxtFNum:   LongInt;  {next free file number}
    ioVFrBlk:     INTEGER); {number of free allocation blocks}
```

```
  cntrlParam:
    . . .  {used by Device Manager}
END;

     VCB = RECORD
             qLink:       QElemPtr;   {next queue entry}
             qType:       INTEGER;    {not used}
             vcbFlags:    INTEGER;    {bit 15=1 if dirty}
             vcbSigWord:  INTEGER;    {always $D2D7}
             vcbCrDate:   LongInt;    {date volume was initialized}
             vcbLsBkUp:   LongInt;    {date of last backup}
             vcbAtrb:     INTEGER;    {volume attributes}
             vcbNmFls:    INTEGER;    {number of files in directory}
             vcbDirSt:    INTEGER;    {directory's first block}
             vcbBlLn:     INTEGER;    {length of file directory}
             vcbNmBlks:   INTEGER;    {number of allocation blocks}
             vcbAlBlkSiz: LongInt;    {size of allocation blocks}
             vcbClpSiz:   LongInt;    {number of bytes to allocate}
             vcbAlBlSt:   INTEGER;    {first block in block map}
             vcbNxtFNum:  LongInt;    {next unused file number}
             vcbFreeBks:  INTEGER;    {number of unused blocks}
             vcbVN:       STRING[27]; {volume name}
             vcbDrvNum:   INTEGER;    {drive number}
             vcbDRefNum:  INTEGER;    {driver reference number}
             vcbFSID:     INTEGER;    {file system identifier}
             vcbVRefNum:  INTEGER;    {volume reference number}
             vcbMAdr:     Ptr;        {location of block map}
             vcbBufAdr:   Ptr;        {location of volume buffer}
             vcbMLen:     INTEGER;    {number of bytes in block map}
             vcbDirIndex: INTEGER;    {used internally}
             vcbDirBlk:   INTEGER     {used internally}
           END;

     DrvQEl = RECORD
                qLink:     QElemPtr; {next queue entry}
                qType:     INTEGER;  {not used}
                dQDrive:   INTEGER;  {drive number}
                dQRefNum:  INTEGER;  {driver reference number}
                dQFSID:    INTEGER;  {file-system identifier}
                dQDrvSize: INTEGER   {number of logical blocks}
              END;
```

High-Level Routines [Pascal only]

Accessing Volumes

```
FUNCTION GetVInfo   (drvNum: INTEGER; volName: StringPtr; VAR
                     vRefNum: INTEGER; VAR freeBytes: LongInt) :
                     OSErr;
FUNCTION GetVol     (volName: StringPtr; VAR vRefNum: INTEGER) :
                     OSErr;
```

```
FUNCTION SetVol     (volName: StringPtr; vRefNum: INTEGER) : OSErr;
FUNCTION FlushVol   (volName: StringPtr; vRefNum: INTEGER) : OSErr;
FUNCTION UnmountVol (volName: StringPtr; vRefNum: INTEGER) : OSErr;
FUNCTION Eject      (volName: StringPtr; vRefNum: INTEGER) : OSErr;
```

Changing File Contents

```
FUNCTION Create   (fileName: Str255; vRefNum: INTEGER; creator:
                   OSType; fileType: OSType) : OSErr;
FUNCTION FSOpen   (fileName: Str255; vRefNum: INTEGER; VAR
                   refNum: INTEGER) :  OSErr;
FUNCTION FSRead   (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr)
                   : OSErr;
FUNCTION FSWrite  (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr)
                   : OSErr;
FUNCTION GetFPos  (refNum: INTEGER; VAR filePos: LongInt) : OSErr;
FUNCTION SetFPos  (refNum: INTEGER; posMode: INTEGER; posOff: LongInt)
                   : OSErr;
FUNCTION GetEOF   (refNum: INTEGER; VAR logEOF: LongInt) : OSErr;
FUNCTION SetEOF   (refNum: INTEGER; logEOF: LongInt) : OSErr;
FUNCTION Allocate (refNum: INTEGER; VAR count: LongInt) : OSErr;
FUNCTION FSClose  (refNum: INTEGER) : OSErr;
```

Changing Information About Files

```
FUNCTION GetFInfo (fileName: Str255; vRefNum: INTEGER; VAR
                   fndrInfo: FInfo) : OSErr;
FUNCTION SetFInfo (fileName: Str255; vRefNum: INTEGER; fndrInfo:
                   FInfo) : OSErr;
FUNCTION SetFLock (fileName: Str255; vRefNum: INTEGER) : OSErr;
FUNCTION RstFLock (fileName: Str255; vRefNum: INTEGER) : OSErr;
FUNCTION Rename   (oldName: Str255; vRefNum: INTEGER; newName:
                   Str255) : OSErr;
FUNCTION FSDelete (fileName: Str255; vRefNum: INTEGER) : OSErr;
```

Low-Level Routines

Initializing the File I/O Queue

```
PROCEDURE InitQueue;
```

Accessing Volumes

```
FUNCTION PBMountVol   (paramBlock: ParmBlkPtr) : OSErr;
FUNCTION PBGetVolInfo (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBGetVol     (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBSetVol     (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBFlshVol    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBUnmountVol (paramBlock: ParmBlkPtr) : OSErr;
FUNCTION PBOffLine    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBEject      (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Changing File Contents

```
FUNCTION PBCreate    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBOpen      (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBOpenRF    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBRead      (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBWrite     (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBGetFPos   (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBSetFPos   (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBGetEOF    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBSetEOF    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBAllocate  (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBFlshFile  (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBClose     (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Changing Information About Files

```
FUNCTION PBGetFInfo (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBSetFInfo (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBSetFLock (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBRstFLock (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBSetFVers (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBRename   (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBDelete   (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Accessing Queues [Pascal only]

```
FUNCTION GetFSQHdr  : QHdrPtr;
FUNCTION GetVCBQHdr : QHdrPtr;
FUNCTION GetDrvQHdr : QHdrPtr;
```

Assembly-Language Information

Constants

```
; Flags in file information used by the Finder

fsQType         .EQU      5     ;I/O request queue entry type
fHasBundle      .EQU      5     ;set if file has a bundle
fInvisible      .EQU      6     ;set if file's icon is invisible

; Flag set when queue is in use

qInUse          .EQU      7     ;set if queue is in use

; Flags for testing trap words

asnycTrpBit     .EQU     1Ø     ;set in trap word for an asynchronous call
noQueueBit      .EQU      9     ;set in trap word for immediate execution
```

Structure of File Information Used by the Finder

```
fdType              File type
fdCreator           File's creator
fdFlags             Flags
fdLocation          File's location
fdFldr              File's window
```

Standard Parameter Block Data Structure

```
qLink               Next queue entry
qType               Queue type
ioTrap              Routine trap
ioCmdAddr           Routine address
ioCompletion        Completion routine
ioResult            Result code
ioFileName          File name (and possibly volume name)
ioVNPtr             Volume name
ioVRefNum           Volume reference number
ioDrvNum            Drive number
```

I/O Parameter Block Data Structure

ioRefNum	Path reference number
ioFileType	Version number
ioPermssn	Read/write permission
ioNewName	New file or volume name for Rename
ioLEOF	Logical end-of-file for SetEOF
ioOwnBuf	Access path buffer
ioNewType	New version number for SetFilType
ioBuffer	Data buffer
ioReqCount	Requested number of bytes
ioActCount	Actual number of bytes
ioPosMode	Newline character and type of positioning operation
ioPosOffset	Size of positioning offset

File Information Parameter Block Data Structure

ioRefNum	Path reference number
ioFileType	Version number
ioFDirIndex	File number
ioFlAttrib	File attributes
ioFFlType	Version number
ioFlUsrWds	Information used by the Finder
ioFFlNum	File number
ioFlStBlk	First allocation block of data fork
ioFlLgLen	Logical end-of-file of data fork
ioFlPyLen	Physical end-of-file of data fork
ioFlRStBlk	First allocation block of resource fork
ioFlRLgLen	Logical end-of-file of resource fork
ioFlRPyLen	Physical end-of-file of resource fork
ioFlCrDat	Date and time file was created
ioFlMdDat	Date and time file was last modified

Volume Information Parameter Block Data Structure

ioVolIndex	Volume index number
ioVCrDate	Date and time volume was initialized
ioVLsBkUp	Date and time of last volume backup
ioVAtrb	Bit 15=1 if volume is locked
ioVNmFls	Number of files in file directory
ioVDirSt	First block of file directory
ioVBlLn	Number of blocks in file directory
ioVNmAlBlks	Number of allocation blocks on volume
ioVAlBlkSiz	Number of bytes per allocation block
ioVClpSiz	Number of bytes to allocate
ioAlBlSt	First block in volume block map
ioVNxtFNum	Next free file number
ioVFrBlk	Number of free allocation blocks

Volume Information Data Structure

drSigWord	Always $D2D7
drCrDate	Date and time of initialization
drLsBkUp	Date and time of last backup
drAtrb	Volume attributes
drNmFls	Number of files in file directory
drDirSt	First logical block of file directory
drBlLen	Number of logical blocks in file directory
drNmAlBlks	Number of allocation blocks on volume
drAlBlkSiz	Size of allocation blocks
drClpSiz	Number of bytes to allocate
drAlBlSt	Logical block number of first allocation block
drNxtFNum	Next unused file number
drFreeBks	Number of unused allocation blocks
drVN	Length and characters of volume name

File Directory Entry Data Structure

flFlags	Bit 7=1 if entry used; bit Ø=1 if file locked
flTyp	Version number
flUsrWds	Information used by the Finder
flFlNum	File number
flStBlk	First allocation block of data fork
flLgLen	Data fork's logical end-of-file
flPyLen	Data fork's physical end-of-file
flRStBlk	First allocation block of resource fork
flRLgLen	Resource fork's logical end-of-file
flRPyLen	Resource fork's physical end-of-file
flCrDat	Date and time file was created
flMdDat	Date and time file was last modified
flName	Length and characters of file name

Queue Header Data Structure

qFlags	Queue flags
qHead	Pointer to first queue entry
qTail	Pointer to last queue entry

Volume Control Block Data Structure

qLink	Next queue entry
qType	Not used
vcbFlags	Bit 15=1 if volume control block is dirty
vcbSigWord	Always $D2D7
vcbCrDate	Date and time volume was initialized
vcbLsBkUp	Date and time last backup copy was made
vcbAtrb	Volume attributes
vcbNmFls	Number of files in directory
vcbDirSt	First logical block of file directory

vcbBlLn	Length of file directory
vcbNmBlks	Number of allocation blocks on volume
vcbAlBlkSiz	Size of allocation blocks
vcbClpSiz	Number of bytes to allocate
vcbAlBlSt	First logical block in block map
vcbNxtFNum	Next unused file number
vcbFreeBks	Number of unused allocation blocks
vcbVN	Length and characters of volume name
vcbDrvNum	Drive number of drive in which volume is mounted
vcbDRefNum	Driver reference number of driver for drive in which volume is mounted
vcbFSID	ID for file system handling volume
vcbVRefNum	Volume reference number
vcbMAdr	Memory location of volume block map
vcbBufAdr	Memory location of volume buffer
vcbMLen	Number of bytes in volume block map
vcbDirIndex	For internal File Manager use
vcbDirBlk	For internal File Manager use

File Control Block Data Structure

fcbFlNum	File number
fcbMdRByt	Flags
fcbTypByt	Version number
fcbSBlk	First allocation block of file
fcbEOF	Logical end-of-file
fcbPLen	Physical end-of-file
fcbCrPs	Mark
fcbVPtr	Location of volume control block
fcbBfAdr	Location of access path buffer
fcbFlPos	For internal use of File Manager

File Control Block Data Structure

qLink	Next queue entry
qType	Always drvType
dQDrive	Drive number
dQRefNum	Driver reference number
dQFSID	File system ID
dQDrvSize	Number of logical blocks

Macro Names

Routine name	Macro name
InitQueue	_InitQueue
PBMountVol	_MountVol
PBGetVolInfo	_GetVolInfo
PBGetVol	_GetVol
PBSetVol	_SetVol
PBFlshVol	_FlushVol
PBUnmountVol	_UnmountVol

PBOffLine	_OffLine
PBEject	_Eject
PBCreate	_Create
PBOpen	_Open
PBOpenRF	_OpenRF
PBRead	_Read
PBWrite	_Write
PBGetFPos	_GetFPos
PBSetFPos	_SetFPos
PBGetEOF	_GetEOF
PBSetEOF	_SetEOF
PBAllocate	_Allocate
PBFlshFile	_FlushFile
PBClose	_Close
PBGetFInfo	_GetFileInfo
PBSetFInfo	_SetFileInfo
PBSetFLock	_SetFilLock
PBRstFLock	_RstFilLock
PBSetFVers	_SetFilType
PBRename	_Rename
PBDelete	_Delete

Variables

Name	Size	Contents
fsQHdr	4 bytes	File I/O queue
vcbQHdr	4 bytes	Volume-control-block queue
defVCBPtr	4 bytes	Pointer to default volume control block
fcbSPtr	4 bytes	Pointer to file-control-block buffer
tagData + 2	4 bytes	Location of file tags
drvQHdr	4 bytes	Drive queue
toExtFS	4 bytes	Pointer to external file system

Result Codes

These values are available as predefined constants in both Pascal and assembly language.

Name	Value	Meaning
badMDBErr	-6Ø	Master directory block is bad; must reinitialize volume
bdNamErr	-37	Bad file name or volume name (perhaps zero-length)
dirFulErr	-33	File directory full
dskFulErr	-34	All allocation blocks on the volume are full
dupFNErr	-48	A file with the specified name already exists
eofErr	-39	Logical end-of-file reached during read operation
extFSErr	-58	External file system; file-system identifier is nonzero, or path reference number is greater than 1Ø24

fBsyErr	-47	One or more files are open
fLckdErr	-45	File locked
fnfErr	-43	File not found
fnOpnErr	-38	File not open
fsRnErr	-59	Problem during Rename
ioErr	-36	Disk I/O error
mFulErr	-41	System heap is full
noErr	Ø	No error
nsDrvErr	-56	Specified drive number doesn't match any number in the drive queue
noMacDskErr	-57	Volume lacks Macintosh-format directory
nsvErr	-35	Specified volume doesn't exist
opWrErr	-49	The read/write permission of only one access path to a file can allow writing
paramErr	-5Ø	Parameters don't specify an existing volume, and there's no default volume
permErr	-54	Read/write permission doesn't allow writing
posErr	-4Ø	Attempted to position before start of file
rfNumErr	-51	Reference number specifies nonexistent access path
tmfoErr	-42	Only 12 files can be open simultaneously
volOffLinErr	-53	Volume not on-line
volOnLinErr	-55	Volume specified is already mounted and on-line
vLckdErr	-46	Volume is locked by a software flag
wrPermErr	-61	Read/write permission or open permission doesn't allow writing
wPrErr	-44	Volume is locked by a hardware setting

GLOSSARY

access path: A description of the route that the File Manager follows to access a file; created when a file is opened.

access path buffer: Memory used by the File Manager to transfer data between an application and a file.

allocation block: Volume space composed of an integral number of logical blocks.

asynchronous execution: During asynchronous execution of a File Manager routine, the calling application is free to perform other tasks.

block map: Same as volume allocation block map.

closed file: A file without an access path. Closed files cannot be read from or written to.

completion routine: Any application-defined code to be executed when an asynchronous call to a File Manager routine is completed.

data buffer: Heap space containing information to be written to a file or driver from an application, or read from a file or driver to an application.

data fork: The part of a file that contains data accessed via the File Manager.

default volume: A volume that will receive I/O during a File Manager routine call, whenever no other volume is specified.

drive number: A number used to identify a disk drive. The internal drive is number 1, and the external drive is number 2.

drive queue: A list of disk drives connected to the Macintosh.

end-of-file: See logical end-of-file or physical end-of-file.

file: A named, ordered sequence of bytes; a principal means by which data is stored and transmitted on the Macintosh.

file control block: 3Ø bytes of system heap space in a file-control-block buffer containing information about an access path.

file-control-block buffer: A 362-byte nonrelocatable block containing one file control block for each access path.

file directory: The part of a volume that contains descriptions and locations of all the files on the volume.

file I/O queue: A queue containing parameter blocks for all I/O requests to the File Manager.

file name: A sequence of up to 255 characters that identifies a file.

file number: A unique number assigned to a file, which the File Manager uses to distinguish it from other files on the volume. A file number specifies the file's entry in a file directory.

file tags: Information associated with each logical block, designed to allow reconstruction of files on a volume whose directory or other file-access information has been destroyed.

fork: One of the two parts of a file; see data fork and resource fork.

I/O request: A request for input from or output to a file or device driver; caused by calling a File Manager or Device Manager routine asynchronously.

locked file: A file whose data cannot be changed.

locked volume: A volume whose data cannot be changed. Volumes can be locked by either a software flag or a hardware setting.

logical block: Volume space composed of 512 consecutive bytes of standard information and an additional number of bytes of disk-driver specific information.

logical end-of-file: The position of one byte past the last byte in a file; equal to the actual number of bytes in the file.

mark: The position of the next byte in a file that will be read or written.

master directory block: Part of the data structure of a volume; contains the volume information and the first 448 bytes of the block map.

mounted volume: A volume that previously was inserted into a disk drive and had descriptive information read from it by the File Manager.

newline character: Any ASCII character, but usually Return (ASCII code $ØD), that indicates the end of a sequence of bytes.

newline mode: A mode of reading data where the end of the data is indicated by a newline character (and not by a specific byte count).

off-line volume: A mounted volume with all but 94 bytes of its descriptive information released.

on-line volume: A mounted volume with its volume buffer and descriptive information contained in memory.

open file: A file with an access path. Open files can be read from and written to.

open permission: Information about a file that indicates whether the file can be read from, written to, or both.

parameter block: Memory space used to transfer information between applications and the File Manager.

path reference number: A number that uniquely identifies an individual access path; assigned when the access path is created.

physical end-of-file: The position of one byte past the last allocation block of a file; equal to 1 more than the maximum number of bytes the file can contain.

read/write permission: Information associated with an access path that indicates whether the file can be read from, written to, both read from and written to, or whatever the file's open permission allows.

resource fork: The part of a file that contains the resources used by an application (such as menus, fonts, and icons) and also the application code itself; usually accessed via the Resource Manager.

synchronous execution: During synchronous execution of a File Manager routine, the calling application must wait until the routine is completed, and isn't free to perform any other task.

unmounted volume: A volume that hasn't been inserted into a disk drive and had descriptive information read from it, or a volume that previously was mounted and has since had the memory used by it released.

version number: A number from Ø to 255 used to distinguish between files with the same name.

volume: A piece of storage medium formatted to contain files; usually a disk or part of a disk. The 3 1/2-inch Macintosh disks are one volume.

volume allocation block map: A list of 12-bit entries, one for each allocation block, that indicate whether the block is currently allocated to a file, whether it's free for use, or which block is next in the file. Block maps exist both on volumes and in memory.

volume attributes: Information contained on volumes and in memory indicating whether the volume is locked, has one or more files open (in memory only), and whether the volume control block matches the volume information (in memory only).

volume buffer: Memory used initially to load the master directory block, and used thereafter for reading from files that are opened without an access path buffer.

volume control block: A 9Ø-byte nonrelocatable block that contains volume-specific information, including the first 64 bytes of the master directory block.

volume-control-block queue: A list of the volume control blocks for all mounted volumes.

volume index: A number identifying a mounted volume listed in the volume-control-block queue. The first volume in the queue has an index of 1, and so on.

volume information: Volume-specific information contained on a volume; includes the volume name, number of files on the volume, and so on.

volume name: A sequence of up to 27 printing characters that identifies a volume; always followed by a colon (:) to distinguish it from a file name.

volume reference number: A unique number assigned to a volume as it's mounted, used to refer to the volume.

COMMENTS?

Macintosh User Education encourages your comments on this manual.

- What do you like or dislike about it?

- Were you able to find the information you needed?

- Was it complete and accurate?

- Do you have any suggestions for improvement?

Please send your comments to the author (indicated on the cover page) at 10460 Bandley Drive M/S 3-G, Cupertino CA 95014. Mark up a copy of the manual or note your remarks separately. (We'll return your marked-up copy if you like.)

Thanks for your help!

MACINTOSH USER EDUCATION

The Device Manager: A Programmer's Guide /DMGR/DEVICE

See Also: The Macintosh User Interface Guidelines
The Memory Manager: A Programmer's Guide
The File Manager: A Programmer's Guide
The Desk Manager: A Programmer's Guide
The Vertical Retrace Manager: A Programmer's Guide
Inside Macintosh: A Road Map
Programming Macintosh Applications in Assembly Language

Modification History: First Draft (ROM 7) Bradley Hacker 6/15/84

ABSTRACT

This manual describes the Device Manager, the part of the Macintosh Operating System that controls the exchange of information between a Macintosh application and devices. It also discusses interrupts.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Device Manager, the part of the Macintosh Operating System that controls the exchange of information between a Macintosh application and devices. It also discusses interrupts. *** Eventually it will become part of the comprehensive Inside Macintosh manual. *** General information about using and writing device drivers can be found in this manual; specific information about the standard Macintosh drivers is contained in separate manuals.

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal. You should also be familiar with the basic concepts behind the Macintosh Operating System's Memory Manager.

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly-language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it.

The manual begins with an introduction to the Device Manager and what you can do with it. It then discusses some basic concepts behind the Device Manager: what devices and device drivers are and how they're used.

A section on using the Device Manager introduces its routines and tells how they fit into the flow of your application. This is followed by detailed descriptions of all the commonly used Device Manager routines, their parameters, calling protocol, effects, side effects, and so on.

Following these descriptions are sections that provide information for programmers who want to write their own drivers, including a discussion of interrupts and a sample device driver.

Finally, there's a summary of the Device Manager, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE DEVICE MANAGER

The Device Manager is the part of the Operating System that handles communication between applications and devices. A device is a part of the Macintosh, or a piece of external equipment, that can transfer information into or out of the Macintosh. Macintosh devices include disk drives, two serial communications ports, the sound generator, and printers. The video screen is **not** a device; drawing on the screen is handled by QuickDraw.

There are two kinds of devices: character devices and block devices. A character device reads or writes a stream of characters, one at a time: it can neither skip characters nor go back to a previous character. A character device is used to get information from or send information to the world outside of the Macintosh Operating System and

memory: it can be an input device, an output device, or an input/output device. The serial ports and printers are all character devices.

A block device reads and writes blocks of 512 characters at a time; it can read or write any accessible block on demand. A block device is usually used to store and retrieve information; disk drives are block devices.

Applications communicate with devices through the Device Manager—either directly, or indirectly through another Operating System or Toolbox "Manager". For example, an application can communicate with a disk drive directly via the Device Manager, or indirectly via the File Manager (which calls the Device Manager). The Device Manager doesn't manipulate devices directly; it calls device drivers that do (Figure 1). Device drivers are programs that take data coming from the Device Manager and convert them into actions of devices, and convert device actions into data for the Device Manager to process.

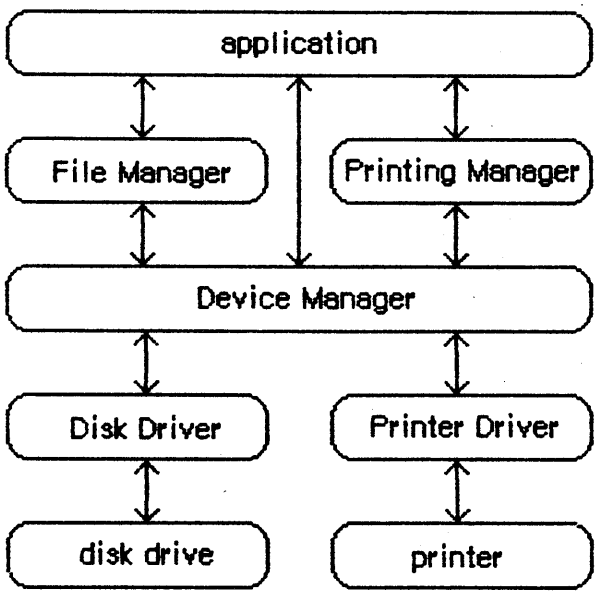


Figure 1. Communication with Devices

The Operating System includes three standard device drivers in ROM: the Disk Driver, the Sound Driver, and the ROM Serial Drivers. There are also a number of standard RAM drivers: the Printer Driver, the RAM Serial Drivers, and desk accessories. RAM drivers are resources, and are read from the system resource file as needed.

You can add other drivers independently or build on top of the existing drivers (for example, the Printer Driver is built on top of the Serial Driver); the section "Writing Your Own Device Drivers" describes how to do this. Desk accessories are a special type of device driver, and are manipulated via the specialized routines of the Desk Manager.

(warning)

Information about desk accessories covered in the Desk Manager manual will not be repeated here. Some information in this manual may not apply to desk accessories.

A device driver can be either open or closed. The Sound Driver and Disk Driver are opened when the system starts up--the rest of the drivers are opened at the specific request of an application. After a driver has been opened, an application can read data from and write data to the driver. You can close device drivers that are no longer in use, and recover the memory used by them. Up to 32 device drivers may be open at any one time.

Before it's opened, you identify a device driver by its driver name; after it's opened, you identify it by its reference number. A driver name consists of a period (.) followed by any sequence of 1 to 254 printing characters. A RAM driver's name is the same as its resource name. You can use uppercase and lowercase letters when naming drivers, but the Device Manager ignores case when comparing names (it doesn't ignore diacritical marks).

(note)

Although device driver names can be quite long, there's little reason for them to be more than a few characters in length.

The Device Manager assigns each open device driver a driver reference number, from -1 to -32, that's used instead of its driver name to refer to it.

Most communication between an application and an open device driver occurs by reading and writing data. Data read from a driver is placed in the application's data buffer, and data written to a driver is taken from the application's data bufffer. A data buffer is memory allocated by the application for communication with drivers.

In addition to data that's read from or written to device drivers, drivers may require or provide other information. Information transmitted to a driver by an application is called control information; information provided by a driver is called status information. Control information may select modes of operation, start or stop processes, enable buffers, choose protocols, and so on. Status information may indicate the current mode of operation, the readiness of the device, the occurrence of errors, and so on. Each device driver may respond to a number of different types of control information and may provide a number of different types of status information.

Each of the standard Macintosh drivers includes predefined calls for transmitting control information and receiving status information. Explanations of these calls can be found in the manuals describing the drivers.

USING THE DEVICE MANAGER

This section discusses how the Device Manager routines for calling device drivers fit into the general flow of an application program and gives an idea of what routines you'll need to use. The routines themselves are described in detail in the section "Device Manager Routines". The Device Manager routines for writing device drivers are described in the section "Writing Your Own Device Drivers"

You can call Device Manager routines via three different methods: high-level Pascal calls, low-level Pascal calls, and assembly language. The high-level Pascal calls are designed for Pascal programmers interested in using the Device Manager in a simple manner; they provide adequate device I/O and don't require much special knowledge to use. The low-level Pascal and assembly-language calls are designed for advanced Pascal programmers and assembly-language programmers interested in using the Device Manager to its fullest capacity; they require some special knowledge to be used most effectively.

(note)

The names used to refer to routines here are actually assembly-language macro names for the low-level routines, but the Pascal routine names are very similar.

The Device Manager is automatically initialized each time the system is started up.

Before an application can exchange information with a device driver, it must open the driver. ROM drivers are opened when the system starts up; for RAM drivers, call Open. The Device Manager will return the driver reference number that you'll use every time you want to refer to that device driver.

An application can send data from its data buffer to an open driver with a Write call, and transfer data from an open driver to its data buffer with Read. An application passes control information to a device driver by calling Control, and receives status information from a driver by calling Status.

Whenever you want to stop a device driver from completing I/O initiated by a Read, Write, Control, or Status call, call KillIO. KillIO halts any current I/O and deletes any pending I/O. For example, you could use KillIO to implement a Cancel button that interrupts printing by your application.

When you're through using a driver, call Close. Close forces the device driver to complete any pending I/O, and then releases all the memory used by the driver.

DEVICE MANAGER ROUTINES

This section describes the Device Manager routines used to call drivers. It's divided into two parts. The first describes all the high-level Pascal routines of the Device Manager, and the second presents information about calling the low-level Pascal and assembly-language routines.

All the Device Manager routines in this section return a result code of type OSErr. Each routine description lists all of the applicable result codes, along with a short description of what the result code means. Lengthier explanations of all the result codes can be found in the summary at the end of this manual.

High-Level Device Manager Routines

The Pascal calls in this section cannot be invoked from assembly language; see the following section for equivalent calls.

(note)
As described in the File Manager manual, the FSRead and FSWrite routines are also used to read from and write to files.

```
FUNCTION OpenDriver (name: Str255; VAR refNum: INTEGER) : OSErr;
```

OpenDriver opens the device driver specified by name and returns its reference number in refNum.

Result codes		
	noErr	No error
	badUnitErr	Bad reference number
	dInstErr	Couldn't find driver in resource file
	openErr	Driver cannot perform the requested reading or writing
	unitEmptyErr	Bad reference number

```
FUNCTION CloseDriver (refNum: INTEGER) : OSErr;
```

CloseDriver closes the device driver having the reference number refNum. Any pending I/O is completed, and the memory used by the driver is released.

Result codes		
	noErr	No error
	badUnitErr	Bad reference number
	dRemoveErr	Tried to remove an open driver
	unitEmptyErr	Bad reference number

```
FUNCTION FSRead (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr) :
          OSErr;
```

FSRead attempts to read the number of bytes specified by the count parameter from the device driver having the reference number refNum, and transfer them to the data buffer pointed to by buffPtr. After the read operation is completed, the number of bytes actually read is returned in the count parameter.

Result codes		
	noErr	No error
	badUnitErr	Bad reference number
	notOpenErr	Driver isn't open
	unitEmptyErr	Bad reference number
	readErr	Driver can't respond to Read calls

```
FUNCTION FSWrite (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr) :
          OSErr;
```

FSWrite attempts to take the number of bytes specified by the count parameter from the buffer pointed to by buffPtr and write them to the open device driver having the reference number refNum. After the write operation is completed, the number of bytes actually written is returned in the count parameter.

Result codes		
	noErr	No error
	badUnitErr	Bad reference number
	notOpenErr	Driver isn't open
	unitEmptyErr	Bad reference number
	writErr	Driver can't respond to Write calls

```
FUNCTION Control (refNum: INTEGER; csCode: INTEGER; csParam: Ptr) :
          OSErr;
```

Control sends control information to the device driver having the reference number refNum. The type of information sent is specified by csCode, and the information itself is pointed to by csParam. The values passed in csCode and pointed to by csParam depend on the driver being called.

Result codes		
	noErr	No error
	badUnitErr	Bad reference number
	notOpenErr	Driver isn't open
	unitEmptyErr	Bad reference number
	controlErr	Driver can't respond to this Control call

```
FUNCTION Status (refNum: INTEGER; csCode: INTEGER; csParam: Ptr) :
          OSErr;
```

Status returns status information about the device driver having the reference number refNum. The type of information returned is specified by csCode, and the information itself is pointed to by csParam. The values passed in csCode and pointed to by csParam depend on the driver being called.

Result codes		
	noErr	No error
	badUnitErr	Bad reference number
	notOpenErr	Driver isn't open
	unitEmptyErr	Bad reference number
	statusErr	Driver can't respond to this Status call

```
FUNCTION KillIO (refNum: INTEGER) : OSErr;
```

KillIO terminates all current and pending I/O with the device driver having the reference number refNum.

Result codes		
	noErr	No error
	badUnitErr	Bad reference number
	unitEmptyErr	Bad reference number
	controlErr	Driver can't respond to KillIO calls

(note)
KillIO is actually a special type of PBControl call, and all information about PBControl calls applies equally to KillIO.

Low-Level Device Manager Routines

This section contains special information for programmers using the low-level Pascal or assembly-language routines of the Device Manager, and then describes the routines in detail.

All low-level Device Manager routines can be executed either synchronously (meaning that the application cannot continue until the I/O is completed) or asynchronously (meaning that the application is free to perform other tasks while the I/O is being completed).

When you call a Device Manager routine asynchronously, an I/O request is placed in the driver's I/O queue, and control returns to the calling application--even before the actual I/O is completed. Requests are taken from the queue one at a time (in the same order that they were entered), and processed. Only one request per driver may be processed at any given time.

The calling application may specify a completion routine to be executed as soon as the I/O operation has been completed.

Routine parameters passed by an application to the Device Manager and returned by the Device Manager to an application are contained in a parameter block, which is memory space in the heap or stack. All low-level Pascal calls to the Device Manager are of the form

```
PBCallName (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

PBCallName is the name of the routine. ParamBlock points to the parameter block containing the parameters for the routine. If async is TRUE, the call is executed asynchronously; if FALSE, it's executed synchronously.

Assembly-language note: When you call a Device Manager routine, AØ must point to a parameter block containing the parameters for the routine. If you want the routine to be executed asynchronously, set bit 1Ø of the routine trap word. You can do this by supplying the word ASYNC as the second argument to the routine macro. For example:

```
_Read   ,ASYNC
```

You can set or test bit 1Ø of a trap word by using the global constant asynTrpBit.

If you want a routine to be executed immediately (bypassing the driver's I/O queue), set bit 9 of the routine trap word. This can be accomplished by supplying the word IMMED as the second argument to the routine macro. (The driver must be able to handle immediate calls for this to work.) For example

```
_Write  ,IMMED
```

You can set or test bit 9 of a trap word by using the global constant noQueueBit. You can specify either ASYNC or IMMED, but not both.

All routines return a result code in DØ.

Routine Parameters

The lengthy, variable-length data structure of a parameter block is given below. The Device Manager and File Manager use this same data structure, but only the parts relevant to the Device Manager are discussed here. Each kind of parameter block contains eight fields of standard information and two to nine fields of additional information:

```
TYPE ParamBlkType = (ioParam, fileParam, volumeParam, cntrlParam);

     ParamBlockRec = RECORD
                       qLink:        QElemPtr;  {next queue entry}
                       qType:        INTEGER;   {queue type}
                       ioTrap:       INTEGER;   {routine trap}
                       ioCmdAddr:    Ptr;       {routine address}
                       ioCompletion: ProcPtr;   {completion routine}
                       ioResult:     OSErr;     {result code}
                       ioNamePtr:    StringPtr; {driver name}
                       ioVRefNum:    INTEGER;   {used by Disk Driver}
                       CASE ParamBlkType OF
                         ioParam:
                         . . .  {I/O routine parameters}
                         fileParam:
                         . . .  {used by File Manager}
                         volumeParam:
                         . . .  {used by File Manager}
                         cntrlParam:
                         . . .  {Control and Status call parameters}
                     END;

     ParmBlkPtr = ^ParamBlockRec;
```

The first four fields in each parameter block are handled entirely by the Device Manager, and most programmers needn't be concerned with them; programmers who are interested in them should see the section "The Structure of a Device Driver".

IOCompletion contains the address of a completion routine to be executed at the end of an asynchronous call; it should be NIL for asynchronous calls with no completion routine, and is automatically set to NIL for all synchronous calls. For asynchronous calls, ioResult is positive while the routine is executing, and returns the result code.

IONamePtr is a pointer to the name of a driver and is used only for calls to the PBOpen routine. IOVRefNum is used by the Disk Driver to identify volumes.

An 8-field parameter block is adequate for opening a driver, but most of the Device Manager routines require longer parameter blocks, as described below.

I/O routines use seven additional fields:

```
ioParam:
  (ioRefNum:    INTEGER;    {driver reference number}
   ioVersNum:   SignedByte; {not used}
   ioPermssn:   SignedByte; {read/write permission}
   ioMisc:      Ptr;        {not used}
   ioBuffer:    Ptr;        {data buffer}
   ioReqCount:  LongInt;    {requested number of bytes}
   ioActCount:  LongInt;    {actual number of bytes}
   ioPosMode:   INTEGER;    {type of positioning operation}
   ioPosOffset: LongInt);   {size of positioning offset}
```

IOPermssn requests permission to read from or write to a driver when the driver is opened, and must contain one of the following predefined constants:

```
fsCurPerm  = Ø; {whatever is currently allowed}
fsRdPerm   = 1; {request to read only}
fsWrPerm   = 2; {request to write only}
fsRdWrPerm = 3; {request to read and write}
```

This request is compared with the capabilities of the driver (some drivers are read-only, some are write-only). If the driver is incapable of performing as requested, an error will be returned.

IOBuffer points to an application's data buffer into which data is written by Read calls and from which data is read by Write calls. IOReqCount specifies the requested number of bytes to be read or written. IOActCount contains the number of bytes actually read or written.

Advanced programmers: IOPosMode and ioPosOffset contain positioning information used for Read and Write calls by drivers of block devices. Bits Ø and 1 of ioPosMode indicate a byte position from the physical beginning of the block-formatted medium (such as a disk); it must contain one of the following predefined constants:

```
fsAtMark    = Ø; {at current position of mark }
                 { (ioPosOffset ignored)}
fsFromStart = 1; {offset relative to beginning of file}
fsFromLEOF  = 2; {offset relative to logical end-of-file}
fsFromMark  = 3; {offset relative to current mark}
```

IOPosOffset specifies the byte offset beyond ioPosMode where the operation is to be performed. Control and Status calls use two additional fields:

```
cntrlParam:
  (csCode:  INTEGER;              {type of Control or Status call}
   csParam: ARRAY[Ø..Ø] OF Byte); {control or status information}
```

CSCode contains a number identifying the type of call. This number may be interpreted differently by each driver. The csParam field contains

the control or status information for the call; it's declared as a zero-length array because its exact contents will vary depending from one Control or Status call to the next.

(note)

Programmers who want to use the low-level Control and Status calls will need to declare their own data type that mimics all fields of the ParamBlockRec except for csParam. For example, if you want to pass a long integer in csParam, declare the following:

```
TYPE MyParamBlockRec = RECORD
                         qLink:   QElemPtr;
                         . . .
                         csCode:  INTEGER;
                         csParam: LongInt;
                       END;

VAR  MyPBR: MyParamBlockRec;
```

Then pass @MyPBR (a pointer to your variable) to the low-level Control and Status routines.

Routine Descriptions

This section describes the procedures and functions. Each routine description includes the low-level Pascal form of the call and the routine's assembly-language macro. A list of the fields in the parameter block affected by the call is also given.

Assembly-language note: The field names given in these descriptions are those of the ParamBlockRec data type; see "Summary of the Device Manager" for the corresponding assembly-language equates.

The number next to each parameter name indicates the byte offset of the parameter from the start of the parameter block pointed to by AØ; only assembly-language programmers need be concerned with it. An arrow drawn next to each parameter name indicates whether it's an input, output, or input/output parameter:

Arrow	Meaning
—→	Parameter is passed to the routine
←—	Parameter is returned by the routine
←→	Parameter is passed to and returned by the routine

(note)
As described in the File Manager manual, the PBOpen and PBClose routines are also used to open and close files.

```
FUNCTION PBOpen (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _Open

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	18	ioNamePtr	pointer
<--	24	ioRefNum	word
-->	27	ioPermssn	byte

Result codes

noErr	No error
badUnitErr	Bad reference number
dInstErr	Couldn't find driver in resource file
openErr	Driver cannot perform the requested reading or writing
unitEmptyErr	Bad reference number

PBOpen opens the device driver specified by ioNamePtr and returns its reference number in ioRefNum. IOPermssn specifies the requested read/write permission.

```
FUNCTION PBClose (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _Close

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word

Result codes

noErr	No error
badUnitErr	Bad reference number
dRemoveErr	Tried to remove an open driver
unitEmptyErr	Bad reference number

PBClose closes the device driver having the reference number ioRefNum. Any pending I/O is completed, and the memory used by the driver is released.

```
FUNCTION PBRead (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _Read

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word
-->	32	ioBuffer	pointer
-->	36	ioReqCount	long word
<--	40	ioActCount	long word
-->	44	ioPosMode	word
<->	46	ioPosOffset	long word

Result codes

noErr	No error
badUnitErr	Bad reference number
notOpenErr	Driver isn't open
unitEmptyErr	Bad reference number
readErr	Driver can't respond to Read calls

PBRead attempts to read ioReqCount bytes from the device driver having the reference number ioRefNum, and transfer them to the data buffer pointed to by ioBuffer. After the read operation is completed, the number of bytes actually read is returned in ioActCount.

Advanced programmers: If the driver is reading from a block device, the byte offset from the position indicated by ioPosMode, where the read should actually begin, is given by ioPosOffset.

```
FUNCTION PBWrite (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _Write

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word
-->	32	ioBuffer	pointer
-->	36	ioReqCount	long word
<--	40	ioActCount	long word
-->	44	ioPosMode	word
-->	46	ioPosOffset	long word

Result codes

noErr	No error
badUnitErr	Bad reference number
notOpenErr	Driver isn't open
unitEmptyErr	Bad reference number
writErr	Driver can't respond to Write calls

PBWrite attempts to take ioReqCount bytes from the buffer pointed to by ioBuffer and write them to the device driver having the reference number ioRefNum. After the write operation is completed, the number of bytes actually written is returned in ioActCount.

Advanced programmers: If the driver is writing to a block device, ioPosMode indicates whether the write should begin relative to the beginning of the device or the current position. The byte offset from the position indicated by ioPosMode, where the write should actually begin, is given by ioPosOffset.

```
FUNCTION PBControl (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _Control

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word
-->	26	csCode	word
-->	28	csParam	record

Result codes

noErr	No error
badUnitErr	Bad reference number
notOpenErr	Driver isn't open
unitEmptyErr	Bad reference number
controlErr	Driver can't respond to this Control call

PBControl sends control information to the device driver having the reference number ioRefNum. The type of information sent is specified by csCode, and the information itself begins at csParam. The values passed in csCode and csParam depend on the driver being called.

```
FUNCTION PBStatus (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _Status

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word
-->	26	csCode	word
-->	28	csParam	record

Result codes

noErr	No error
badUnitErr	Bad reference number
notOpenErr	Driver isn't open
unitEmptyErr	Bad reference number
statusErr	Driver can't respond to this Status call

PBStatus returns status information about the device driver having the reference number ioRefNum. The type of information returned is specified by csCode, and the information itself begins at csParam. The values passed in csCode and csParam depend on the driver being called.

```
FUNCTION PBKillIO (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Trap macro _KillIO

Parameter block

-->	12	ioCompletion	pointer
<--	16	ioResult	word
-->	24	ioRefNum	word

Result codes

noErr	No error
badUnitErr	Bad reference number
unitEmptyErr	Bad reference number
controlErr	Driver can't respond to KillIO calls

KillIO stops any current I/O request being processed, and removes all pending I/O requests from the I/O queue of the device driver having the reference number ioRefNum. The completion routine of each pending I/O request is called, with ioResult equal to the following result code:

```
CONST abortErr = -27;
```

(note)
KillIO is actually a special type of Control call, and all information about Control calls applies equally to KillIO.

THE STRUCTURE OF A DEVICE DRIVER

This section describes the structure of device drivers for programmers interested in writing their own driver or manipulating existing drivers. Most of the information presented here is accessible only through assembly language.

RAM drivers are stored in resource files. The resource type for drivers is 'DRVR'. The resource name is the driver name. The resource ID for a driver is its unit number (explained below) and will be between Ø and 31 inclusive. Don't use the unit number of an existing driver unless you want the existing driver to be replaced.

As illustrated in Figure 2, a driver begins with a few words of flags and other data, followed by offsets to the routines that do the work of the driver, an optional title, and finally the routines themselves.

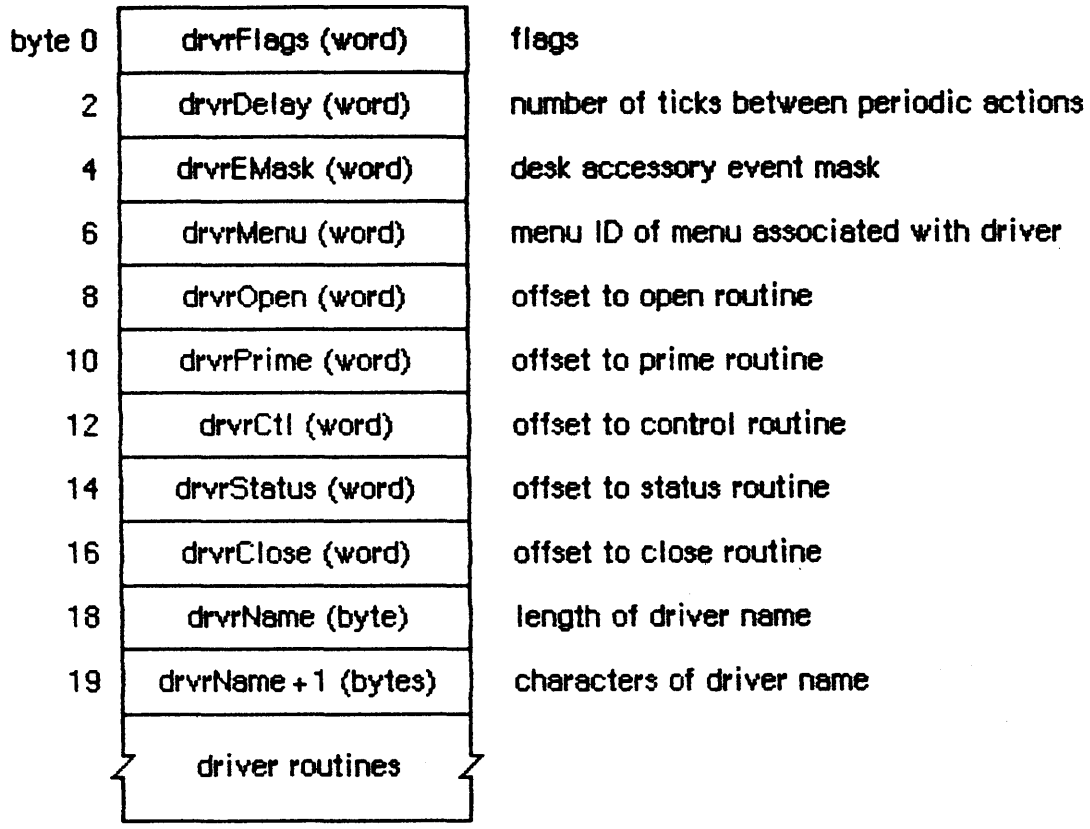


Figure 2. Driver Structure

Every driver contains a routine to handle Open and Close calls, and may contain routines to handle Read, Write, Control, Status, and KillIO calls. The driver routines that handle Device Manager calls are as follows:

Device Manager call	Driver routine
Open	Open
Read	Prime
Write	Prime
Control	Control
KillIO	Control
Status	Status
Close	Close

For example, when a KillIO call is made to a driver, the driver's control routine must implement the call. Each bit of the **high-order**

bytes of the drvrFlags word contains a flag:

```
dReadEnable     .EQU    Ø   ;set if driver can respond to Read calls
dWritEnable     .EQU    1   ;set if driver can respond to Write calls
dCtlEnable      .EQU    2   ;set if driver can respond to Control calls
dStatEnable     .EQU    3   ;set if driver can respond to Status calls
dNeedGoodBye    .EQU    4   ;set if driver needs to be called before the
                            ; application heap is reinitialized
dNeedTime       .EQU    5   ;set if driver needs time for performing a
                            ; periodic action
dNeedLock       .EQU    6   ;set if driver will be locked in memory as
                            ; soon as it's opened (always set for
                            ; ROM drivers)
```

Bits 8 through 11 indicate which Device Manager calls the driver's routines can respond to.

Unlocked RAM drivers that exist on the application heap will be lost every time the heap is reinitialized (when an application starts up, for example). If dNeedGoodBye is set, the control routine of the device driver will be called before the heap is reinitialized, and the driver can perform any "clean-up" actions it needs to. The driver's control routine identifies this "good-bye" call by checking the csCode parameter--it will be -1.

Device drivers may need to perform predefined actions periodically. For example, a network driver may want to poll its input buffer every ten seconds to see if it has received any messages. If the dNeedTime flag is set, the driver **does** need to perform a periodic action, and the drvrDelay word contains a tick count indicating how often the periodic action should occur. A tick count of Ø means it should happen as often as possible, 1 means it should happen every 6Øth of a second, 2 means every 3Øth of a second, and so on. Whether the action actually occurs this frequently depends on how often you call the Desk Manager routine SystemTask. SystemTask calls the driver's control routine (if the time indicated by drvrDelay has elapsed), and the control routine must perform whatever predefined action is desired. The driver's control routine identifies the SystemTask call by checking the csCode parameter--it will be the global constant accRun.

(note)

> Some drivers may not want to rely on the application to call SystemTask, and should install their own task in the vertical retrace queue to accomplish the desired action (see the Vertical Retrace Manager manual).

DrvrEMask and drvrMenu are used only for desk accessories and are discussed in the Desk Manager manual.

Following drvrMenu are the offsets to the driver routines, a title for the driver (preceded by its length in bytes), and the routines that do the work of the driver.

A Device Control Entry

The first time a driver is opened, information about it is read into a structure in memory called a device control entry. A device control entry tells the Device Manager the location of the driver's routines, the location of the driver's I/O queue, and other information. A device control entry is a 40-byte relocatable block located on the system heap. It's locked while the driver is open, and unlocked while the driver is closed.

The structure of a device control entry is illustrated in Figure 3. Notice that some of the data is taken from the first four words of the driver. Most of the data in the device control entry is stored and accessed only by the Device Manager, but in some cases the driver itself must store into it.

byte 0	dCtlDriver (long word)	pointer to ROM driver or handle to RAM driver
4	dCtlFlags (word)	flags
6	dCtlQueue (word)	low-order byte: driver's version number
8	dCtlQHead (pointer)	pointer to first entry in driver's I/O queue
12	dCtlQTail (pointer)	pointer to last entry in driver's I/O queue
16	dCtlPosition (long word)	byte position used by Read and Write calls
20	dCtlStorage (handle)	handle to RAM driver's private storage
24	dCtlRefNum (word)	driver's reference number
26	dCtlCurTicks (long word)	used internally by Device Manager
30	dCtlWindow (pointer)	pointer to driver's window record (if any)
34	dCtlDelay (word)	number of ticks between periodic actions
36	dCtlEMask (word)	desk accessory event mask
38	dCtlMenu (word)	menu ID of menu associated with driver

Figure 3. Device Control Entry

The low-order byte of the dCtlFlags word contains the following flags:

```
dOpened         .EQU    5   ;set if driver is open
dRAMBased       .EQU    6   ;set if driver is RAM-based
drvrActive      .EQU    7   ;set if driver is currently executing
```

The high-order byte contains information copied from the drvrFlags word of the driver:

```
dReadEnable     .EQU    Ø   ;set if driver can respond to Read calls
dWritEnable     .EQU    1   ;set if driver can respond to Write calls
dCtlEnable      .EQU    2   ;set if driver can respond to Control calls
dStatEnable     .EQU    3   ;set if driver can respond to Status calls
dNeedGoodBye    .EQU    4   ;set if driver needs to be called before the
                            ; application heap is reinitialized
dNeedTime       .EQU    5   ;set if driver needs time for performing a
                            ; periodic action
dNeedLock       .EQU    6   ;set if driver will be locked in memory as
                            ; soon as it's opened (always set for
                            ; ROM drivers)
```

DCtlPosition is used only by drivers of block devices, and indicates the current source or destination position of a Read or Write call. The position is given as a number of bytes beyond the physical beginning of the medium used by the device. For example, if one logical block of data has just been read from a 3 1/2-inch disk via the Disk Driver, dCtlPosition will be 512.

ROM drivers generally use locations in low memory for their local storage. RAM drivers may reserve memory within their code space, or allocate a relocatable block and keep a handle to it in dCtlStorage (if the block resides in the application heap, its handle will be set to NIL when the heap is reinitialized).

The Unit Table

The location of each device control entry is maintained in a list called the unit table. The unit table is a 128-byte nonrelocatable block containing 32 4-byte entries. Each entry has a number, from Ø to 31, called the unit number, and contains a handle to the device control entry for a driver. The unit number can be used as an index into the unit table to locate the handle to a specific driver's device control entry; it's equal to

```
-1 * (refNum + 1)
```

where refNum is the driver's reference number. For example, the Sound Driver's reference number is -4 and its unit number is 3.

Figure 4 shows the layout of the unit table just after the system starts up.

(note)

Any new drivers contained in resource files should have resource IDs that don't conflict with the unit numbers of existing drivers--unless you want an existing driver to be replaced.

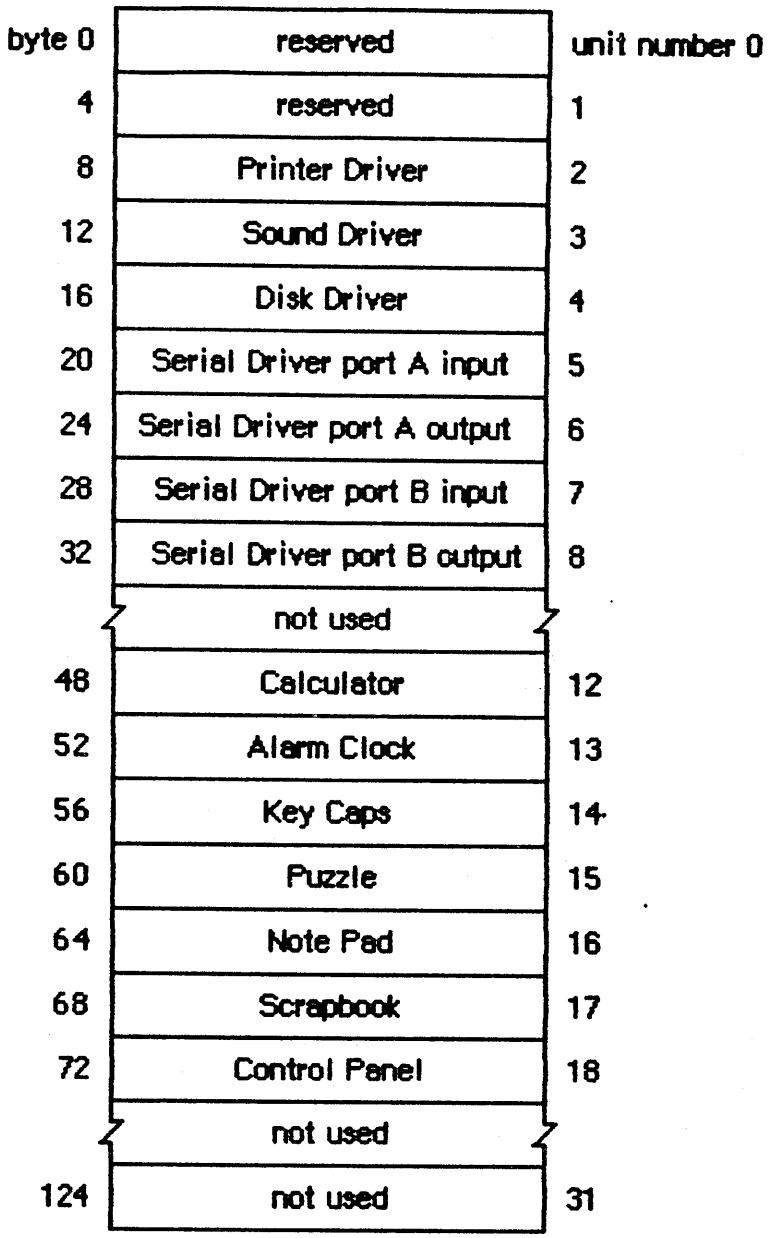


Figure 4. The Unit Table

Assembly-language note: The global variable uTableBase points to the unit table.

Each device driver contains an I/O queue with a list of I/O requests to be completed by the driver. A driver I/O queue is a standard Operating

System queue (described in the Operating System Utilities manual *** doesn't yet exist; for now, see the appendix of the File Manager manual ***). The queue is located in the device control entry for the driver (Figure 5).

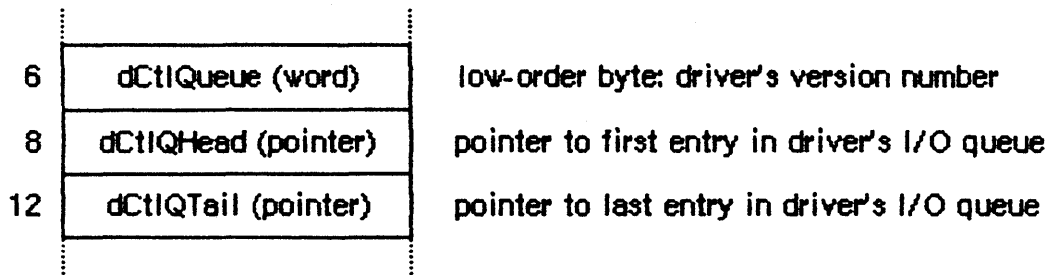


Figure 5. Driver I/O Queue Structure

The three fields shown in Figure 5 are analogous to the QHdr data type of a standard Operating System queue.

Each driver I/O queue uses entries of type ioQType. Each entry in the queue consists of a parameter block for the routine that was called. The structure of this block is shown in part below:

```
TYPE ParamBlockRec = RECORD
                       qLink:     QElemPtr; {next queue entry}
                       qType:     INTEGER;  {queue type}
                       ioTrap:    INTEGER;  {routine trap}
                       ioCmdAddr: Ptr   ;   {routine address}
                         . . .              {rest of block}
                     END;
```

QLink points to the next entry in the queue, and qType indicates the queue type, which must always be ORD(ioQType). IOTrap and ioCmdAddr contain the trap and address of the Device Manager routine that was called. You can use the following global constants to identify Device Manager traps, by comparing the global constant with the low-order byte of the trap:

```
aRdCmd      .EQU    2     ;Read call (trap $AØØ2)
aWrCmd      .EQU    3     ;Write call (trap $AØØ3)
aCtlCmd     .EQU    4     ;Control call (trap $AØØ4)
aStsCmd     .EQU    5     ;Status call (trap $AØØ5)
```

You can get a pointer to a driver's I/O queue by calling the Device Manager function GetDCtlQHdr.

```
FUNCTION GetDCtlQHdr (refNum: INTEGER) : QHdrPtr;  [Pascal only]
```

GetDCtlQHdr returns a pointer to the I/O queue of the device driver having the reference number refNum.

Assembly-language note: To access the contents of a driver's I/O queue from assembly language, you can use offsets from the address of the global variable dCtlQueue.

WRITING YOUR OWN DEVICE DRIVERS

This section describes what you'll need to do to write your own device driver. If you aren't interested in writing your own driver, skip ahead to the summary.

Drivers are usually written in assembly language. The structure of your driver must match that shown in the previous section. The routines that do the work of the driver should be written to operate the device in whatever way you require. Your driver must contain routines to handle Open and Close calls, and may choose to handle Read, Write, Control, Status, and KillIO calls as well.

When the Device Manager executes a driver routine to handle an application call, it passes a pointer to the call's parameter block in AØ and a pointer to the driver's device control entry in A1. From this information, the driver can determine exactly what operations are required to fulfill the call's requests, and do them.

Open and close routines must execute synchronously. They needn't preserve any registers that they use. Open and close routines should place a result code in DØ and return via an RTS instruction. *** Currently the Device Manager sets DØ to zero upon return from an Open call. ***

The open routine must allocate any private storage required by the driver, store a handle to it in the device control entry (in the dCtlStorage field), initialize any local variables, and then be ready to receive a Read, Write, Status, Control, or KillIO call. It might also install interrupt handlers, change interrupt vectors, and store a pointer to the device control entry somewhere in its local storage for its interrupt handlers to use. The close routine must reverse the effects of the open routine, by releasing all used memory, removing interrupt handlers, and replacing changed interrupt vectors. If anything about the operational state of the driver should be saved until the next time the driver is opened, it should be kept in the relocatable block of memory pointed to by dCtlStorage.

Prime, control, and status routines must be able to respond to queued calls and asynchronous calls, and should be interrupt-driven. Asynchronous portions of the routines can use registers AØ to A3 and DØ to D3, but must preserve any other registers used; synchronous portions can use all registers. Prime, control, and status routines should

return a result code in DØ. They must return via an RTS if called immediately (with IMMED as the second argument to the routine macro) or via an RTS if the device couldn't complete the I/O request right away, or via a JMP to the IODone routine (explained below) if the device completed the request.

(warning)
If they can be called as the result of an interrupt, the prime, control, and status routines should never call Memory Manager routines that cause heap compactions.

The prime routine must implement all Read and Write calls made to the driver. It can distinguish between Read and Write calls by checking the value of the ioTrap field. You may want to use the Fetch and Stash routines described below to read and write characters. If the driver is for a block device, it should update the dCtlPosition field of the device control entry after each read or write. The control routine must accept the control information passed to it, and manipulate the device as requested. The status routine must return requested status information. Since both the control and status routines may be subjected to Control and Status calls sending and requesting a variety of information, they must be prepared to respond correctly to all types. The control routine must handle KillIO calls; the driver identifies KillIO calls by checking the csCode parameter--it will be the global constant killCode.

(warning)
KillIO calls must return via an RTS, and shouldn't jump (via JMP) to the IODone routine.

Routines for Writing Drivers

The Device Manager includes three routines, Fetch, Stash, and IODone, that provide low-level support for driver routines. Include them in the code of your device driver if they're useful to you. Fetch, Stash, and IODone are invoked via "jump vectors" (jFetch, jStash, and jIODone) rather than macros (in the interest of speed). You use a jump vector by moving its address onto the stack:

```
MOVE.L     jIODone,-(SP)
RTS
```

Fetch and Stash don't return a result code, since the only result possible is dSIOCoreErr, which invokes the System Error Handler. IODone can return a result code.

Fetch Function

Jump vector	jFetch
On entry	A1: pointer to device control entry
On exit	DØ: character fetched; bit 15=1 if it's the last character in the data buffer

Fetch gets the next character from the data buffer pointed to by ioBuffer and places it in DØ. IOActCount is incremented by 1. If ioActCount equals ioReqCount, bit 15 of DØ is set. After receiving the last byte requested, the driver should call IODone.

Stash Function

Jump vector	jStash
On entry	A1: pointer to device control entry DØ: character to stash
On exit	DØ: bit 15=1 if it's the last character requested

Stash places the character in DØ into the data buffer pointed to by ioBuffer, and increments ioActCount by 1. If ioActCount equals ioReqCount, bit 15 of DØ is set. After stashing the last byte requested, the driver should call IODone.

IODone Function

Jump vector	jIODone	
On entry	A1: pointer to device control entry	
On exit	DØ: result code	
Result codes	noErr	No error
	unitEmptyErr	Reference number specifies NIL handle in unit table

IODone removes the current I/O request from the driver's I/O queue, marks the driver inactive, unlocks the driver and its device control entry (if it's allowed to by the dNeedLock bit of the dCtlFlags word), and executes the completion routine (if there is one). Then it begins executing the next I/O request in the I/O queue.

A Sample Driver

Here's the skeleton of the Disk Driver, as an example of how a driver should be constructed.

```
; Driver header

DiskDrvr
                .WORD   $4FØØ                   ;RAM driver, read, write,
                                                ; control, status, needs
                                                ; lock
                .WORD   Ø,Ø                      ;no delay or event mask
                .WORD   Ø                       ;no menu

; Offsets to driver routines

                .WORD   DiskOpen-DiskDrvr       ;open
                .WORD   DiskPrime-DiskDrvr      ;prime
                .WORD   DiskControl-DiskDrvr    ;control
                .WORD   DiskStatus-DiskDrvr     ;status
                .WORD   allDone-DiskDrvr        ;close (just RTS)
                .BYTE   5                       ;length of name
                .ASCII  '.Disk'                 ;driver name

; Local variables and constants

; Driver routines

; Open routine

DiskOpen        MOVEQ   #<DiskVarLth/2>,DØ      ;get memory for variables
                . . .                           ;allocate variables
                . . .                           ;initialize drive queue
                . . .                           ;install a vertical-
```

```
                                                ; retrace task
DiskRTS         RTS

DiskDone        MOVE.W   DØ,DskErr              ;return result code
                MOVE.L   DiskVars,A1            ;get pointer to locals
                CLR.B    Active(A1)             ;driver isn't active
                MOVE.L   DiskUnitPtr(A1),A1     ;return pointer to DCE
                MOVE.L   jIODone,-(SP)          ;go to IODone
                RTS

; Prime routine

DiskPrime       ORI      #$Ø1ØØ, SR             ;exclude vertical-retrace
                                                ; interrupts
                . . .
                RTS

; Control routine
;               AØ (input): pointer to Control call's parameter block
;               csCode = killCode for KillIO, ejectCode for Eject

DiskControl     MOVE.W     csCode(AØ),DØ        ;get the control code
                SUBQ.W     #killCode,DØ         ;is it KillIO?
                BNE.S      @Ø                   ;branch if not
                MOVE       SR,-(SP)
                ORI        #$Ø1ØØ,SR            ;no VIA interrupts
                BSR        PowerDown            ;start power down,
                                                ; get VIA address
                MOVE.B     #$2Ø,VIER(A2)        ;remove any pending
                                                ; timer interrupts
                RTE                             ;special for KillIO

@Ø              SUBQ.W     #<ejectCode-killCode>,DØ   ;Eject?
                BEQ.S      @1                   ;branch if so
                MOVEQ      #controlErr,DØ       ;can't handle csCode
                BRA.S      DiskDone             ;exit

@1              BSR.S      CkDrvNum             ;set drive to eject
                . . .
                BRA.S      DiskDone             ;exit

; Status routine
;               AØ (input): pointer to Status call's parameter block
;               csCode = 8 for drive status

DiskStatus      MOVE.Q     #statusErr,DØ        ;assume status error

                CMP.W      #drvrStsCode,csCode(AØ) ;drive status call?
                BNE.S      DiskDone             ;exit for other calls

                BSR.S      CkDrvNum
                BNE.S      DiskDone             ;exit on error

                .END
```

Interrupts

This section discusses interrupts: how the Macintosh uses them, and how you can use them if you're writing your own device driver. Only programmers who want to write their own interrupt-driven device drivers need read this section. Programmers who want to build their own driver on top of a built-in Macintosh driver may be interested in some of the information presented here.

An interrupt is a form of exception: an error or abnormal condition detected by the processor in the course of program execution. Specifically, an interrupt is an exception that's signaled to the processor by a device, as distinct from a trap, which arises directly from the execution of an instruction. Interrupts are used by devices to notify the processor of a change in condition of the device, such as the completion of an I/O request. An interrupt causes the processor to suspend normal execution, save the address of the next instruction and the processor's internal status on the stack, and execute an interrupt handler.

The MC68ØØØ recognizes seven different levels of interrupt, each with its own interrupt handler. The addresses of the various handlers, called interrupt vectors, are kept in a vector table in the system communication area. Each level of interrupt has its own vector located in the vector table. When an interrupt occurs, the processor fetches the proper vector from the table, uses it to locate the interrupt handler for that level of interrupt, and jumps to the handler. On completion, the handler exits with an RTE instruction, which restores the internal state of the processor from the stack and resumes normal execution from the point of suspension.

There are three devices that can create interrupts: the 6522 Versatile Interface Adapter (VIA), the 853Ø Serial Communications Controller, and the debugging switch. They send a 3-bit number, from Ø to 7, called the interrupt priority level, to the processor. The interrupt level indicates which device is interrupting, and indicates which interrupt handler should be executed:

Level	Interrupting device
Ø	None
1	VIA
2	SCC
3	VIA and SCC
4-7	Debugging button

A level-3 interrupt occurs when both the VIA and SCC interrupt at the same instant; the interrupt handler for a level-3 interrupt is simply an RTE instruction. Debugging interrupts shouldn't occur during the normal execution of an application.

The interrupt priority level is compared with the processor priority in bits 8, 9, and 1Ø of the status register. If the interrupt priority level is greater than the processor priority, the MC68ØØØ acknowledges the interrupt and initiates interrupt processing. The processor priority determines which interrupting devices are ignored, and which are serviced:

Level	Services
Ø	All interrupts
1	VIA and debugging interrupts only
2	SCC and debugging interrupts only
3-6	Debugging interrupts only
7	No interrupts

When an interrupt is acknowledged, the processor priority is set to the interrupt priority level, to prevent additional interrupts of equal or lower priority, until the interrupt handler has finished servicing the interrupt.

The interrupt priority level is used as an index into the primary interrupt vector table. This table contains seven long words beginning at address $64. Each long word contains the starting address of an interrupt handler (Figure 6).

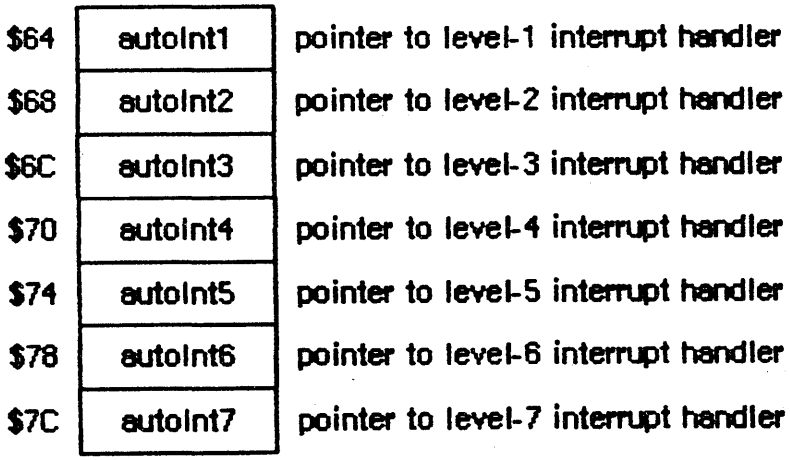


Figure 6. Primary Interrupt Vector Table

Execution jumps to the interrupt handler at the address specified in the table. The interrupt handler then must identify and service the interrupt. Then, it must restore the processor priority, status register, and program counter to the values they contained before the interrupt occurred.

Level-1 (VIA) Interrupts

Level-1 interrupts are generated by the VIA. You'll need to read the Synertek manual describing the VIA to use most of the information provided in this section. The level-1 interrupt handler determines the

source of the interrupt (via the VIA's IFR and IER registers) and then uses a table of secondary vectors in the system communication area to determine which interrupt handler to call (Figure 7).

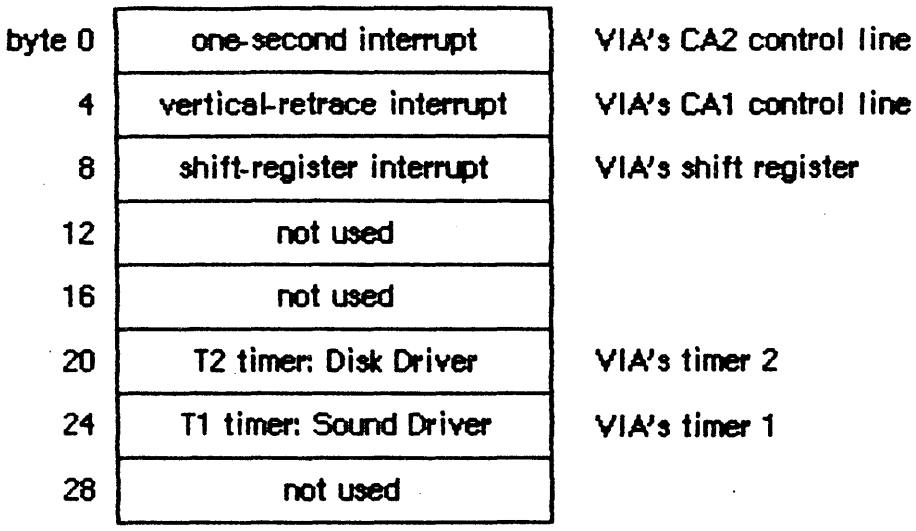


Figure 7. Level-1 Secondary Interrupt Vector Table

The level-1 secondary interrupt vector table begins at the address of the global variable lvl1DT. Each vector in the table points to the interrupt handler for a different source of interrupt. The interrupts are handled in order of their entry in the table, and only one interrupt handler is called per level-1 interrupt (even if two or more sources are interrupting). This allows the level-1 interrupt handler to be reentrant, and interrupt handlers should lower the processor priority as soon as possible in order to enable other pending interrupts to be processed.

One-second interrupts occur every second, and simply update the system global variable time (explained in the Operating System Utilities manual *** doesn't yet exist ***) and invert menu items that are chosen. Vertical retrace interrupts are generated once every vertical retrace interval; control is passed to the Vertical Retrace Manager, which updates the global variable named ticks, handles changes in the state of the cursor, keyboard, and mouse button, and executes tasks installed in the vertical retrace queue.

The shift-register interrupt is used by the Keyboard/Mouse Handler. Whenever the Disk Driver or Sound Driver isn't being used, you can use the T1 and T2 timers for your own needs.

If the cumulative elapsed time for all tasks during a vertical retrace interrupt exceeds 16 milliseconds (one video frame), the vertical retrace interrupt may itself be interrupted by another vertical retrace interrupt. In this case, the second vertical retrace interrupt is ignored.

The base address of the VIA (stored in the global variable VIA) is passed to each interrupt handler in A1.

Level-2 (SCC) Interrupts

Level-2 interrupts are generated by the SCC. You'll need to read the Zilog manual describing the SCC to effectively use the information provided in this section. The level-2 interrupt handler determines the source of the interrupt, and then uses a table of secondary vectors in the system communication area to determine which interrupt handler to call (Figure 8).

byte 0	channel B transmit buffer empty	
4	channel B external/status change	mouse vertical
8	channel B receive character available	
12	channel B special receive condition	
16	channel A transmit buffer empty	
20	channel A external/status change	mouse horizontal
24	channel A receive character available	
28	channel A special receive condition	

Figure 8. Level-2 Secondary Interrupt Vector Table

The level-2 secondary interrupt vector table begins at the address of the global variable lvl2DT. Each vector in the table points to the interrupt handler for a different source of interrupt. The interrupts are handled according to the following fixed priority:

channel A receive character available and special receive
channel A transmit buffer empty
channel A external/status change
channel B receive character available and special receive
channel B transmit buffer empty
channel B external/status change

Only one interrupt handler is called per level-2 interrupt (even if two or more sources are interrupting). This allows the level-2 interrupt handler to be reentrant, and interrupt handlers should lower the processor priority as soon as possible in order to enable other pending interrupts to be processed.

External/status interrupts pass through a tertiary vector table in the system communication area to determine which interrupt handler to call (Figure 9).

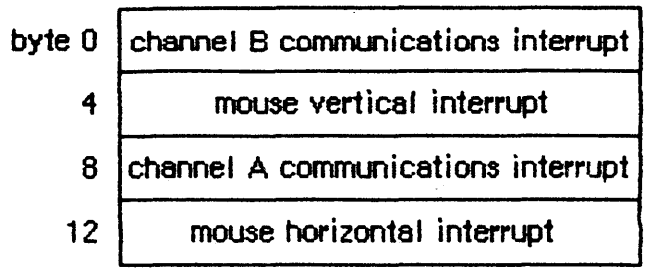


Figure 9. Level-2 External/Status Interrupt Vector Table

The external/status interrupt vector table begins at the address of the global variable extStsDT. Each vector in the table points to the interrupt handler for a different source of interrupt. Communications interrupts (break/abort, for example) are always handled before mouse interrupts.

When a level-2 interrupt handler is called, DØ contains the address of the SCC read register Ø (external/status interrupts only), and D1 contains the bits of read register Ø that have changed since the last external/status interrupt. AØ points to the SCC channel A or channel B control read address and A1 points to SCC channel A or channel B control write address, depending on which channel is interrupting. The SCC's data read address and data write address are located four bytes beyond AØ and A1, respectively. The following global constants can be used to refer to these locations:

Global constant	Value	Refers to
bCtl	Ø	Offset for channel B control
aCtl	2	Offset for channel A control
bData	4	Offset for channel B data
aData	6	Offset for channel A data

Writing Your Own Interrupt Handlers

You can write your own interrupt handlers to replace any of the standard interrupt handlers just described. Be sure to place a vector that points to your interrupt handler in one of the vector tables.

Both the level-1 and level-2 interrupt handlers preserve AØ through A3 and DØ through D3. Every interrupt handler (except for external/status interrupt handlers) is responsible for clearing the source of the interrupt, and for saving and restoring any additional registers used. Interrupt handlers should return directly via an RTS instruction, unless the interrupt is handled immediately, in which case they should jump (via JMP) to the IODone routine.

SUMMARY OF THE DEVICE MANAGER

Constants

```
{ Values for posMode and ioPosMode }

CONST fsAtMark    = Ø; {at current position of mark }
                       { (ioPosOffset ignored)}
      fsFromStart = 1; {offset relative to beginning of file}
      fsFromLEOF  = 2; {offset relative to logical end-of-file}
      fsFromMark  = 3; {offset relative to current mark}

{ Values for requesting read/write access }

      fsCurPerm  = Ø; {whatever is currently allowed}
      fsRdPerm   = 1; {request to read only}
      fsWrPerm   = 2; {request to write only}
      fsRdWrPerm = 3; {request to read and write}
```

Data Types

```
TYPE ParmBlkPtr    = ^ParamBlockRec;

     ParamBlkType  = (ioParam, fileParam, volumeParam, cntrlParam);

     ParamBlockRec = RECORD
        qLink:         QElemPtr;  {next queue entry}
        qType:         INTEGER;   {queue type}
        ioTrap:        INTEGER;   {routine trap}
        ioCmdAddr:     Ptr;       {routine address}
        ioCompletion:  ProcPtr;   {completion routine}
        ioResult:      OSErr;     {result code}
        ioNamePtr:     StringPtr; {driver name}
        ioVRefNum:     INTEGER;   {used by Disk Driver}
     CASE ParamBlkType OF
      ioParam:
       (ioRefNum:     INTEGER;    {driver reference number}
        ioVersNum:    SignedByte; {not used}
        ioPermssn:    SignedByte; {read/write permission}
        ioMisc:       Ptr;        {not used}
        ioBuffer:     Ptr;        {data buffer}
        ioReqCount:   LongInt;    {requested number of bytes}
        ioActCount:   LongInt;    {actual number of bytes}
        ioPosMode:    INTEGER;    {type of positioning operation}
        ioPosOffset:  LongInt);   {size of positioning offset}
      fileParam:
        . . . {used by File Manager}
      volumeParam:
        . . . {used by File Manager}
```

```
    cntrlParam:
     (csCode:  INTEGER;              {type of Control or Status call}
      csParam: ARRAY[Ø..Ø] OF Byte); {control or status information}
    END;
```

High-Level Routines

```
FUNCTION OpenDriver  (name: Str255; VAR refNum: INTEGER) : OSErr;
FUNCTION CloseDriver (refNum: INTEGER) : OSErr;
FUNCTION FSRead  (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr)
                  : OSErr;
FUNCTION FSWrite (refNum: INTEGER; VAR count: LongInt; buffPtr: Ptr)
                  : OSErr;
FUNCTION Control (refNum: INTEGER; csCode: INTEGER; csParam: Ptr) :
                  OSErr;
FUNCTION Status  (refNum: INTEGER; csCode: INTEGER; csParam: Ptr) :
                  OSErr;
FUNCTION KillIO  (refNum: INTEGER) : OSErr;
```

Low-Level Routines

```
FUNCTION PBOpen    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBClose   (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBRead    (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBWrite   (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBControl (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBStatus  (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
FUNCTION PBKillIO  (paramBlock: ParmBlkPtr; async: BOOLEAN) : OSErr;
```

Accessing a Driver's I/O Queue

```
FUNCTION GetDCtlQHdr (refNum: INTEGER) : QHdrPtr;
```

Assembly-Language Information

Constants

```
; I/O queue type

ioQType         .EQU     2        ;I/O request queue entry type

; Driver flags

dReadEnable     .EQU     Ø   ;set if driver can respond to Read calls
dWritEnable     .EQU     1   ;set if driver can respond to Write calls
dCtlEnable      .EQU     2   ;set if driver can respond to Control calls
dStatEnable     .EQU     3   ;set if driver can respond to Status calls
dNeedGoodBye    .EQU     4   ;set if driver needs to be called before the
```

```
                                ; application heap is reinitialized
dNeedTime       .EQU    5       ;set if driver needs time for performing a
                                ; periodic action
dNeedLock       .EQU    6       ;set if driver will be locked in memory as
                                ; soon as it's opened (always set for
                                ; ROM drivers)

; Device control entry flags

dOpened         .EQU    5       ;set if driver is open
dRAMBased       .EQU    6       ;set if driver is RAM-based
drvrActive      .EQU    7       ;set if driver is currently executing

; Trap words for Device Manager calls

aRdCmd      .EQU    2       ;Read call (trap $AØØ2)
aWrCmd      .EQU    3       ;Write call (trap $AØØ3)
aCtlCmd     .EQU    4       ;Control call (trap $AØØ4)
aStsCmd     .EQU    5       ;Status call (trap $AØØ5)

; Offsets for SCC

bCtl      .EQU    Ø     ;Offset for SCC channel B control
aCtl      .EQU    2     ;Offset for SCC channel A control
bData     .EQU    4     ;Offset for SCC channel B data
aData     .EQU    6     ;Offset for SCC channel A data
```

Standard Parameter Block Data Structure

```
qLink               Next queue entry
qType               Queue type
ioTrap              Routine trap
ioCmdAddr           Routine address
ioCompletion        Completion routine
ioResult            Result code
ioFileName          File name (and possibly volume name too)
ioVNPtr             Volume name
ioVRefNum           Volume reference number
ioDrvNum            Drive number
```

Control and Status Parameter Block Data Structure

```
csCode              Type of Control or Status call
csParam             Parameters for Control or Status call
```

I/O Parameter Block Data Structure

ioRefNum	Driver reference number
ioFileType	Not used
ioPermssn	Open permission
ioBuffer	Data buffer
ioReqCount	Requested number of bytes
ioActCount	Actual number of bytes
ioPosMode	Type of positioning operation
ioPosOffset	Size of positioning offset

Driver Structure

drvrFlags	Flags
drvrDelay	Number of ticks between periodic actions
drvrEMask	Desk accessory event mask
drvrMenu	Menu ID of menu associated with driver
drvrOpen	Offset to open routine
drvrPrime	Offset to prime routine
drvrCtl	Offset to control routine
drvrStatus	Offset to status routine
drvrClose	Offset to close routine
drvrName	Length and characters of driver name

Device Control Entry Data Structure

dCtlDriver	Pointer to ROM driver or handle to RAM driver
dCtlFlags	Flags
dCtlQueue	Low-order byte is driver's version number
dCtlQHead	Pointer to first entry in driver's I/O queue
dCtlTail	Pointer to last entry in driver's I/O queue
dCtlPosition	Byte position used by Read and Write calls
dCtlStorage	Handle to RAM driver's private storage
dCtlRefNum	Driver's reference number
dCtlCurTicks	Used internally by Device Manager
dCtlWindow	Pointer to driver's window record (if any)
dCtlDelay	Number of ticks between periodic actions
dCtlEMask	Desk accessory event mask
dCtlMenu	Menu ID of menu associated with driver

Primary Interrupt Vector Table

autoInt1	Pointer to level-1 interrupt handler
autoInt2	Pointer to level-2 interrupt handler
autoInt3	Pointer to level-3 interrupt handler
autoInt4	Pointer to level-4 interrupt handler
autoInt5	Pointer to level-5 interrupt handler
autoInt6	Pointer to level-6 interrupt handler
autoInt7	Pointer to level-7 interrupt handler

I/O Parameter Block Data Structure

ioRefNum Driver reference number

Macro Names

Routine name	Macro name
PBRead	_Read
PBWrite	_Write
PBControl	_Control
PBStatus	_Status
PBKillIO	_KillIO

Routines for Writing Drivers

Routine	Jump vector
Fetch	jFetch
Stash	jStash
IODone	jIODone

Variables

Name	Size	Contents
uTableBase	4 bytes	Pointer to unit table
unitNtryCnt	2 bytes	Maximum number of entries in unit table
lvl1DT	4 bytes	Beginning of level-1 secondary interrupt vector table
lvl2DT	4 bytes	Beginning of level-2 secondary interrupt vector table
extStsDT	4 bytes	Beginning of external/status interrupt vector table
sccRBase	4 bytes	SCC base read address
sccWBase	4 bytes	SCC base write address
VIA	4 bytes	VIA base address

Result Codes

Name	Value	Meaning
abortErr	-27	I/O request aborted by KillIO
badUnitErr	-21	Reference number doesn't match unit table
controlErr	-17	Driver can't respond to this Control call
dInstErr	-26	Couldn't find driver in resource file
dRemoveErr	-25	Tried to remove an open driver
noErr	Ø	No error
notOpenErr	-28	Driver isn't open
openErr	-23	Requested read/write permission doesn't match driver's open permission
readErr	-19	Driver can't respond to Read calls
statusErr	-18	Driver can't respond to this Status call
unitEmptyErr	-22	Reference number specifies NIL handle in unit table
writErr	-2Ø	Driver can't respond to Write calls

GLOSSARY

asynchronous execution: After calling a routine asynchronously, an application is free to perform other tasks until the routine is completed.

block device: A device that reads and writes blocks of 512 characters at a time; it can read or write any accessible block on demand.

character device: A device that reads or writes a stream of characters, one at a time: it can neither skip characters nor go back to a previous character.

closed driver: A device driver that cannot be read from or written to.

close routine: The part of a driver's code that implements Device Manager Close calls.

completion routine: Any application-defined code to be executed when an asynchronous call to a Device Manager routine is completed.

control information: Information transmitted by an application to a device driver; it can typically select modes of operation, start or stop processes, enable buffers, choose protocols, and so on.

control routine: The part of a device driver's code that implements Device Manager Control and KillIO calls.

data buffer: Heap space containing information to be written to a file or driver from an application, or read from a file or driver to an application.

device: A part of the Macintosh or a piece of external equipment, that can transfer information into or out of the Macintosh.

device control entry: A 40-byte relocatable block of heap space that tells the Device Manager the location of a driver's routines, the location of a driver's I/O queue, and other information.

device driver: A program that exchanges information between an application and a device.

driver name: A sequence of up to 254 printing characters used to refer to an open device driver; driver names always begin with a period (.).

driver reference number: A number that uniquely identifies an individual device driver.

exception: An error or abnormal condition detected by the processor in the course of program execution.

interrupt: An exception that's signaled to the processor by a device, to notify the processor of a change in condition of the device, such as

the completion of an I/O request.

interrupt handler: A routine that services interrupts.

interrupt priority level: A number identifying the importance of the interrupt. It indicates which device is interrupting, and which interrupt handler should be executed.

interrupt vector: A pointer to an interrupt handler.

I/O queue: A queue containing the parameter blocks of all I/O requests for one driver.

I/O request: A request for input from or output to a file or device driver; caused by calling a File Manager or Device Manager routine asynchronously.

open driver: A driver that can be read from and written to.

open routine: The part of a device driver's code that implements Device Manager Open calls.

parameter block: An area of heap space used to transfer information between applications and the Device Manager.

prime routine: The part of a device driver's code that implements Device Manager Read and Write calls.

processor priority: Bits 8, 9, and 1Ø of the MC68ØØØ's status register, that indicate which interrupts will be processed and which will be ignored.

status information: Information transmitted to an application by a device driver; it may indicate the current mode of operation, the readiness of the device, the occurrence of errors, and so on.

status routine: The part of a device driver's code that implements Device Manager Status calls.

synchronous execution: After calling a routine synchronously, an application cannot continue execution until the routine is completed.

unit number: The number of each device driver's entry in the unit table.

unit table: A 128-byte nonrelocatable block containing a handle to the device control entry for each device driver.

vector: A pointer.

vector table: A table of vectors in the system communication area.

COMMENTS?

Macintosh User Education encourages your comments on this manual.

- What do you like or dislike about it?

- Were you able to find the information you needed?

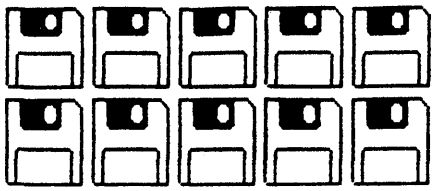
- Was it complete and accurate?

- Do you have any suggestions for improvement?

Please send your comments to the author (indicated on the cover page) at 10460 Bandley Drive M/S 3-G, Cupertino CA 95014. Mark up a copy of the manual or note your remarks separately. (We'll return your marked-up copy if you like.)

Thanks for your help!

Macintosh Sony Disk Driver
Programmer Documentation
8 May 84, LAK



This note is intended for programmers who wish to access the internal/external sony drives directly, bypassing the Macintosh file system. The sony drives are controlled by a ROM driver, surprisingly named 'the sony driver' or 'the sony driver in ROM'. The file system always accesses the sony drives via this driver, and this documentation simply documents that driver interface. The reader should be familiar with whatever generic driver documentation is currently available before reading this. This document is also geared toward assembly language programmers, altho a Pascal interface to generic driver calls is available and may be used. The reasons for accessing the disk driver directly instead of via the file system include writing disk copy software, copy protection software, other file systems, disk test software, other disk drivers, special demo software, or simply for fun as an idle exercise.

Macintosh ROM drivers include the sony driver (.Sony), the sound driver (.Sound), and the asynchronous serial communications driver (.AIn, .AOut, .BIn, .BOut). The sony driver is opened at boot[1] time (system boot is done via a read call to the sony driver which reads the first two disk blocks, called the boot blocks, off the disk into memory) and is never closed. In fact, if a Close call to the driver is made, the system will not function normally thereafter. Since the driver is opened at boot time it is not necessary to open it before using it (Open calls should not affect the driver). The remaining routines which are used to access the disk are Read, Write, Control, and Status.

Control calls

KillIO	CSCode=1 Current I/O is aborted (this call has probably never been used . . .)
Eject	IODrvNum=1 (internal drive) or 2 (external drive) CSCode=7 Any diskette in the specified drive is ejected. This call is made asynchronously from the file system.
Set Tag Buffer	CSCode=8 CSParam: contains a longword pointer to a buffer to be used to get (writes) or put (reads) disk block tag information. Zero defaults to no separate tag buffer:

the 12-byte tag information for the last block read or written is available in the low memory area TagData+2. Disk copy programs use this disk driver feature to preserve disk tag information (see below).

Status

Drive Status

IODrvNum=1 (internal drive) or 2 (external drive)
CSCode=8
CSParam: returns 22 bytes of drive status
(00) - current track
(02) - bit 7 = 1 = write-protected - this is the only byte looked at by the Mac file system . . .
(03) - $FC-$FF = just ejected
0 = no disk-in-place, 1,2 = disk-in-place,
(04) - 0 = may be installed, 1 = drive installed,
$FF = drive not installed
(05) - bit 7 = 0 = single-sided drive
(06) - drive queue element
(18) - $FF for 2-sided format this diskette (valid when byte 03 = 2)
(19) - $FF if prime routine has been called
(20) - word soft error count

Read, Write:

IODrvNum=1 (internal drive) or 2 (external drive)
IOPosMode+1=$00 - cuurent position
$01 - absolute
$03 - relative to current position
$4x - read-verify mode
IOPosOffset= byte position (should be a 512-byte multiple)
IOBuffer=pointer to read/write buffer (may be on any byte boundary)
IOByteCount= bytes to read/write (should be 512 mult)
TagData+2= tag info for first block when writing with no tag buffer installed (the disk driver increments the file block number for multiple block reads).
The file system always reads/writes using absolute addressing mode and uses TagData+2 info to preserve current block tags and write new ones. The file system supports read-verify mode using the disk driver's read-verify mode for whole blocks. Byte address 0 corresponds to sector 0, track 0, side 0. A single-sided diskette contains 800 512-byte sectors.

Disk block tags

Each disk block (sector) is actually 524 bytes long which consists of a 12-byte block tag and 512 bytes of data. The file system works in concert with the disk driver to create and preserve meaningful tags. These tags are not actually used by the system software but are meant to be data which a disk scavenge program would use to recreate the directory of a trashed disk. The 12 bytes consist of the following fields:

(00) - unique file number of file which owns this block
(04) - file flags word (used to differentiate resource/regular fork blocks, and determine last file version byte used
(06) - relative block number of this block in the file
(08) - longword timestamp (from system global Time) set when this file block was written (except when a separate file tag buffer is being used).

Drive Queue

When the Sony driver is opened, it installs two drive queue elements into the system drive queue, one for both the internal and external sony drives controlled by this driver (the drive queue element for the external drive is later dequeued if that drive is not installed). This queue is used by the system to bind drives to drivers to file systems. The definition for a drive queue element is somewhat non-standard:

(00) write-protect: bit 7 = 1 = write-protected
(01) disk in place: $FC-$FF=just ejected, 0=none, 1,2=disk in place, 8=non-ejectable disk in place
(02) installed: 0=unknown, 1=installed, $FF=no disk (sony driver only)
(03) sides: bit 7 = 0 = single-sided (sony driver only)
(04) link pointer
(08) flags (unused)
(10) drive number (1 and 2 are reserved for the int and ext drives)
(12) driver refnum (of driver which is used to access this drives)
(14) file system ID of file system owning this drive (0 for Mac FS)
(16) number of 512-byte blocks on this drive (undefined for sony drives)

Formatting

Due to ROM code size limitations, the current sony driver has no formatting capability (missing the crucial write address mark routine); this is accomplished via the disk format package which may be viewed as logically part of this driver. That package is documented elsewhere.

Driver Hooks

The sony driver accesses main low-level routines via low-memory vectors; this was done to allow access at these low levels for test programs, disk format packages, bug fixes, program debug and performance analysis code, and for certain copy-protection schemes. There are 15 such vectors, defined in the file SONYEQU.TEXT:

JFigTrkSpeed	- routine to determine current disk speed
JDiskPrime	- all calls to disk prime go thru this hook (useful for performance measurement, debug)
JRdAddr	- read address mark routine
JRdData	- read data mark routine (copy protection, test hook)
JWrData	- write data mark routine (copy protection, test hook)
JSeek	- seek to a track routine
JSetUpPoll	- sets up for serial port polling
JRecal	- recal routine (currently patched in RAM)
JControl	- all calls to disk control go thru this vector
JWakeUp	- disk timer wakeup routine (uses VIA timer 2)
JReSeek	- another disk prime hook
JMakeSpdTbl	- routine which constructs speed table
JAdrDisk	- used by external test, format programs only
JSetSpeed	- used by external test, format programs only
NiblTbl	- used by external test, format programs only

These low-level routines will probably be documented here, eventually.

Some Examples

Ejecting the diskette in drive 1:

```
            .INCLUDE    SYSEQU.TEXT             ; relevant equates

MyEject     MOVEQ       #<IOQElSize/2>-1,D0     ; first clear an IO
@1          CLR.W       -(SP)                   ;  parameter block off
            DBRA        D0,@1                   ;  the stack (zeroed)
            MOVE.L      SP,A0                   ; A0 points to it

            MOVE.W      #DskRfn,IORefNum(A0) ; disk driver refnum
            MOVE.W      #1,IODrvNum(A0)      ; drive 1 is internal drive
            MOVE.W      #EjectCode,CSCode(A0) ; eject control code
```

```
            _Eject                          ; do it synchronously

            ADD         #IOQElSize,SP       ; clean up the stack
                                            ; D0 contains a result code
```

Reading block 4 from the diskette in drive 1, asynchronously:

```
            .INCLUDE    SYSEQU.TEXT         ; relevant equates

MyRead      MOVEQ       #<IOQElSize/2>-1,D0 ; first clear an IO
@1          CLR.W       -(SP)               ;  parameter block off
            DBRA        D0,@1               ;  the stack (zeroed)
            MOVE.L      SP,A0               ; A0 points to it

            MOVE.W      #DskRfn,IORefNum(A0) ; disk driver refnum
            MOVE.W      #1,IODrvNum(A0)     ; drive 1 is internal drive
            MOVE.W      #1,IOPosMode(A0)    ; absolute positioning
            MOVE.L      #<512*4>,IOPosOffset(A0) ; block 4 byte position
            MOVE.L      #512,IOByteCount(A0) ; read one block's worth
            LEA         MyBuffer,A1
            MOVE.L      A1,IOBuffer(A0)     ; set up buffer address
            _Read       ,ASYNC              ; do it asynchronously

;           do any other processing here . . . then when the block is needed:

@2          MOVE.W      IOResult(A0),D0     ; wait for completion
            BGT.S       @2

            ADD         #IOQElSize,SP       ; clean up the stack
                                            ; D0 contains a result code

MyBuffer    .BLOCK      512,0               ; sector buffer (should really
                                            ;  come off the heap or stack)
```

---

[1] Boot time is the time between when a Macintosh is powered on and the first application program (usually the Finder) is launched. Boot is short for bootstrap which is from the phrase "pulling oneself up from one's bootstraps" which is really impossible due to physical laws but seems plausible to small children watching cartoons. The first two disk blocks contain the code which is needed to finishing initializing the Macintosh system software and start the first application (the name of which is contained in those blocks).

MACINTOSH USER EDUCATION

The Sound Driver: A Programmer's Guide /DEVICE/SOUND

See Also: The Macintosh User Interface Guidelines
Macintosh Operating System Manual
The Device Manager: A Programmer's Guide

Modification History: First Draft (ROM 7) B. Hacker 3/nn/84

*** Preliminary Draft. Not for distribution *** ABSTRACT

The Sound Driver is a set of data types and routines in the Macintosh Operating System for handling sound and music generation in a Macintosh application. This manual describes the Sound Driver in detail.

TABLE OF CONTENTS

ABOUT THIS MANUAL

The Sound Driver is a set of data structures and routines in the Macintosh Operating System for handling sound and music generation in a Macintosh application. This manual describes the Sound Driver in detail. *** Eventually it will become part of a larger manual describing the entire Toolbox and Operating System. ***

(note)

This manual describes the Sound Driver in version 7 of the ROM. If you're using a different version, the information presented here may not apply.

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal. You should also be familiar with the following:

- the basic concepts behind the Macintosh Operating System's Memory Manager

- devices and device drivers, as described in the Device Manager Manual *** doesn't yet exist ***

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it. *** Currently a Pascal interface to the Sound Driver doesn't exist ***

The manual begins with an introduction to the Sound Driver and what you can do with it. It then steps back a little and looks at the mathematical and physical concepts that form the foundation for the Sound Driver: waveforms, wave frequency, wave amplitude, and wave periods. Once you understand these concepts, read on about how they're translated into sound, music, and speech.

Next, a section on using the Sound Driver describes how you can use Device Manager calls in your application to produce desired sounds. This includes a detailed description of the Sound Driver's control routine—its parameters, calling protocol, effects, and so on.

Finally, there's a summary of the Sound Driver data structures and routine calls, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE SOUND DRIVER

The Sound Driver is a standard Macintosh device driver used to synthesize sound waves. You can use the Sound Driver to generate sound characterized by any kind of waveform by using the three different sound synthesizers in the Sound Driver:

- The four-tone synthesizer is used to make simple harmonic tones, with up to four "voices" producing sound simultaneously; it requires about 50% of the microprocessor's attention during any given time interval.

- The square-wave synthesizer is used to produce less harmonic sounds such as beeps, and requires about 2% of the processor's time.

- The free-form synthesizer is used to make complex music and speech; it requires about 20% of the processor's time.

Figure 1 depicts the waveform of a typical sound wave, and the terms used to describe it. The amplitude is the vertical distance between any given point on the wave and the horizontal line about which the amplitude oscillates; you can think of the amplitude of a wave as its volume level. The wavelength is the horizontal extent of one complete cycle of the wave. Both the amplitude and wavelength can be measured in any unit of distance. The period is the time elapsed during one complete cycle of a wave. The frequency is the reciprocal of the period, or the number of cycles per second (also called Hertz). The phase is some fraction of a wave cycle (measured from a fixed point on the wave).

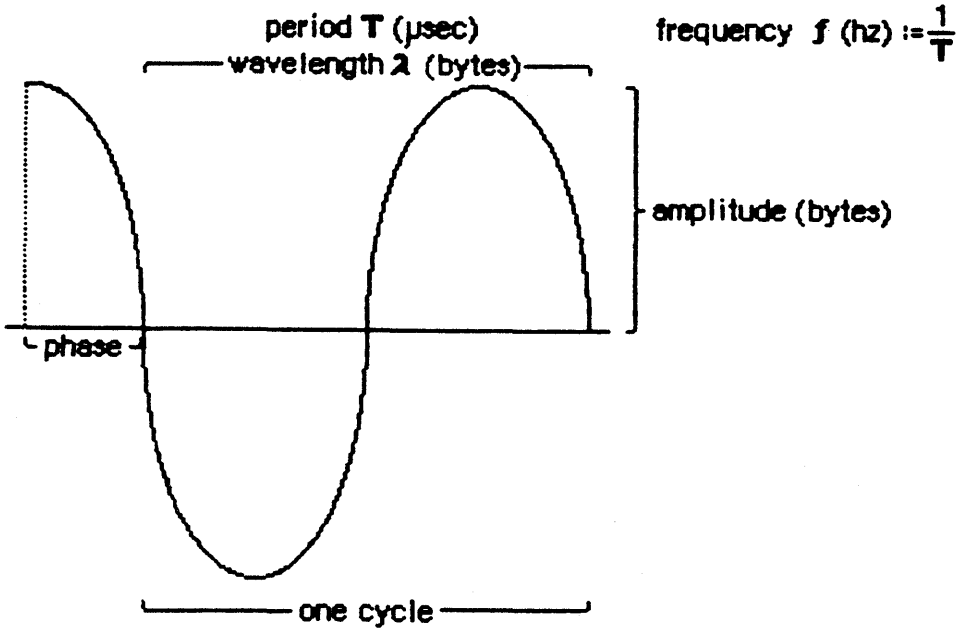


Figure 1. A Waveform

There are many different types of waveforms, three of which are depicted in Figure 2. Sine waves are generated by objects that oscillate periodically at a single frequency (such as a guitar string). Square waves are generated by objects that toggle instantly between two states at a single frequency (such as a doorbell buzzer). Free-form waves are the most common waves of all, and are generated by all

objects that vibrate at rapidly changing frequencies with rapidly changing amplitudes (such as your vocal cords or the instruments of an orchestra all playing at once).

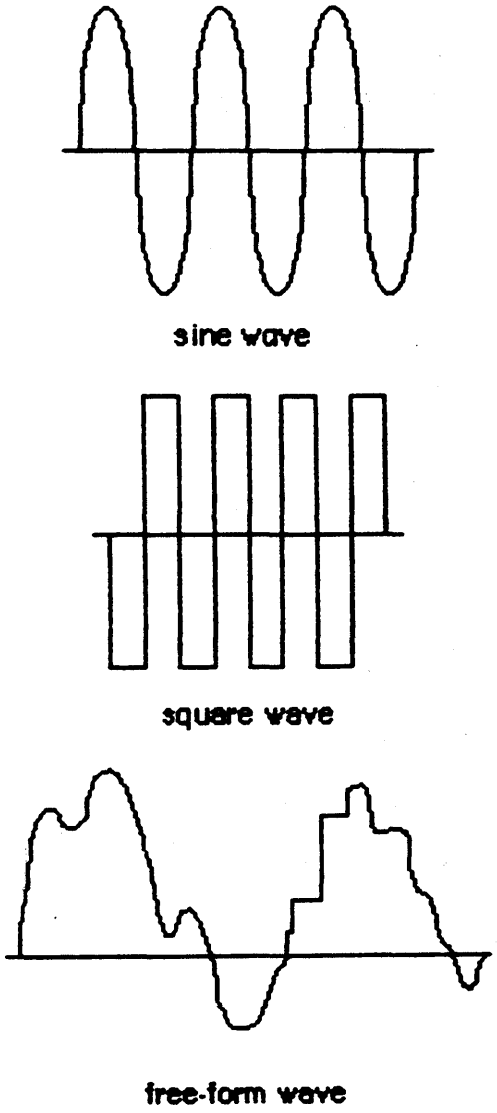


Figure 2. Types of Waveforms

Figure 3 shows the analog representation of a waveform. The Sound Driver represents waveforms digitally, so all waveforms must be converted from their analog representation to a digital representation. The rows of numbers at the bottom of the figure are digital representations of the waveform. The numbers in the upper row are the amplitudes relative to the horizontal zero-amplitude line. The numbers in the lower row all represent the same relative amplitudes, but have been normalized to positive numbers.

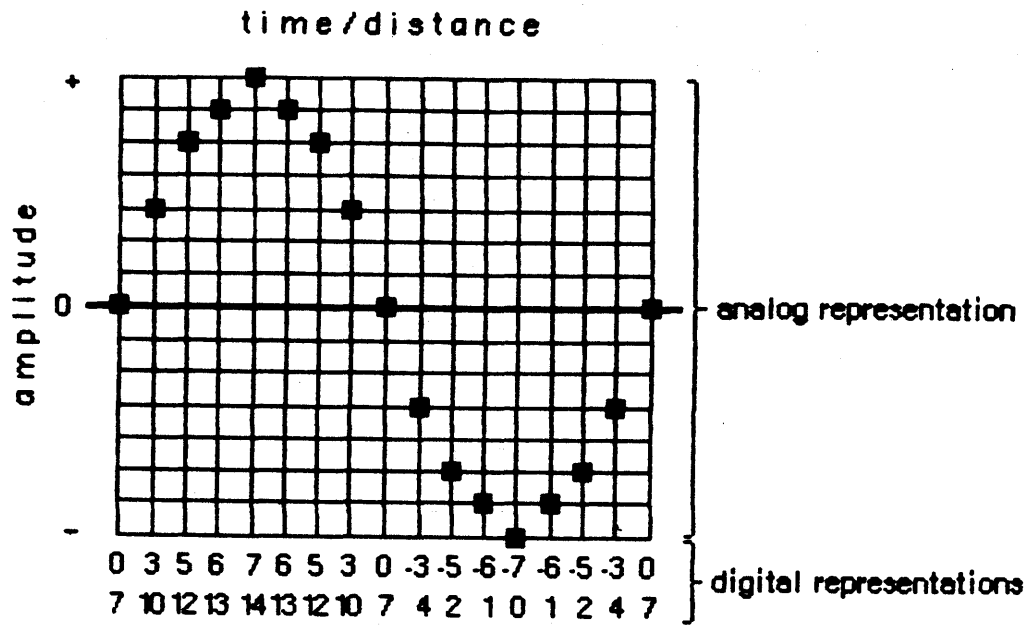


Figure 3. Analog and Digital Representations of a Waveform

A digital representation of a waveform is simply a sequence of wave amplitudes measured at fixed intervals. This sequence of amplitudes is stored in the Sound Driver as a sequence of bytes, each one of which specifies an instantaneous voltage to be sent to the speaker. The bytes are stored in a data structure called a waveform description. Since a sequence of bytes can only represent a group of numbers whose maximum and minimum values differ by less than 256, the amplitudes of your waveforms must be constrained to these same limits.

SOUND DRIVER SYNTHESIZERS

A description of the sound to be generated by a synthesizer is contained in a data structure called a synthesizer buffer. A synthesizer buffer contains the duration, pitch, phase, and waveform of the sound the synthesizer will generate. The exact structure of a synthesizer buffer differs for each type of synthesizer being used.

Free-Form Synthesizer

The free-form synthesizer is used to synthesize complex music and speech. The sound to be produced is represented as a waveform whose complexity and length are limited only by available memory.

A free-form synthesizer buffer consists of one integer and one long integer followed by a waveform description (Figure 4). The waveform description can contain up to 256 bytes. Each amplitude in the waveform description will be generated once; when the end of the

waveform is reached, the synthesizer will stop. The integer must be Ø, to identify the buffer as a free-form buffer. The duration long integer determines the length of time (in 44.93 usec increments) each amplitude in the waveform will be produced. The high-order word of the duration long integer contains the integral part and the low-order word contains the fractional part of the duration. (Binary fractions are described in the Toolbox Utilities manual under Fixed-Point Numbers.)

The time interval specified by the duration long integer can vary between 44.93 usec and 2.95 sec, corresponding to the binary fractions 1.ØØØØ (represented by the four bytes $ØØ Ø1 ØØ ØØ, or the long integer 1) and 65535.9999 (represented by the four bytes $FF FF FF FF, or the long integer 4294967295), respectively.

(note)

As a further example, the time interval 89.86 usec corresponds to the binary fraction 2.ØØØØ, the four bytes $ØØ Ø2 ØØ ØØ, and the long integer 131Ø72. The time interval .Ø115 sec corresponds to the binary fraction 25.5ØØØ, the four bytes $ØØ 19 8Ø ØØ, and the long integer 1671168.

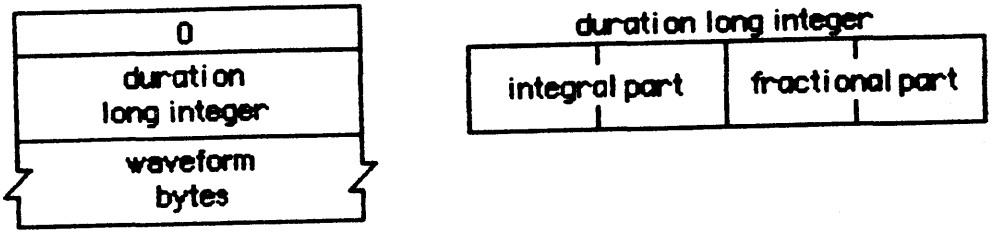


Figure 4. Free-Form Synthesizer Buffer

(note)

Note that the duration long integer specifies a time interval, but it doesn't specify the period of a wave cycle. To determine the time period of a wave cycle in the waveform, use the following relationship:

period = time interval * wavelength

where the wavelength is given in bytes. For example, the period of a wave of 1Ø-byte wavelength with a time interval of 2 usec/byte would be 9ØØ usec (corresponding to 1111 Hz).

Assembly-language note: The address of the free-form buffer currently in use is contained in the system global soundBase.

Square-Wave Synthesizer

The square-wave synthesizer is used to make sounds such as beeps. A square-wave synthesizer buffer consists of a negative integer followed by a sequence of integer triplets (Figure 5). The negative integer identifies the buffer as a square-wave buffer. Each triplet contains the count, amplitude, and duration of a different sound. The square-wave synthesizer doesn't require a waveform description because of the simple form of square waves. You can store as many triplets in a synthesizer buffer as there's room for.

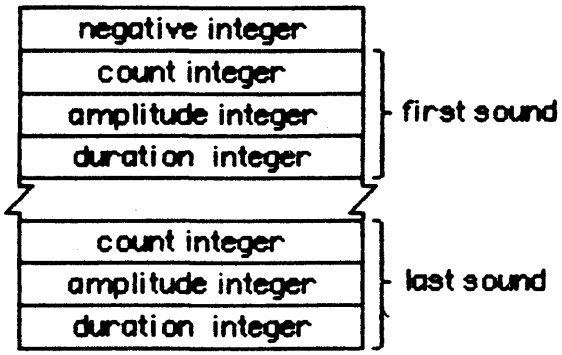


Figure 5. Square-Wave Synthesizer Buffer

Each count integer can range in value from Ø to 65535; the actual frequency the count corresponds to is given by the relationship:

frequency (Hz) = 783360 / count

A partial list of count values and corresponding frequencies for notes comprising Ptolemy's diatonic scale (the scale to which pianos are tuned) is given in the summary at the end of this manual.

Assembly-language note: The value of count currently in use is contained in the system global curPitch.

Each amplitude integer can range from Ø to 255. Each duration integer specifies the number of ticks the sound will be generated, ranging from Ø to 255 (corresponding to Ø to 4.25 seconds).

The last sound triplet must be signified by a count integer of Ø. When the square-wave synthesizer is used, the sound specified by each triplet is generated once, and then the synthesizer stops.

Four-Tone Synthesizer

The four-tone synthesizer is used to produce harmonic sounds such as music. It can simultaneously generate four different sounds, each with its own frequency, phase, and waveform.

A four-tone synthesizer buffer consists of an integer and a pointer (Figure 6). The integer can be any positive number, and serves only to identify the buffer as a four-tone buffer. The pointer points to a data structure describing the four tones, called a four-tone record.

Assembly-language note: The address of the four-tone record currently in use is stored in the system global soundPtr.

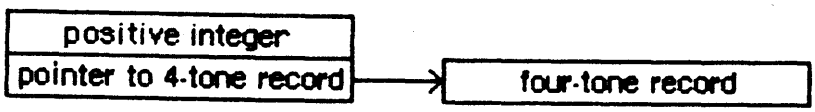


Figure 6. Four-Tone Synthesizer Buffer

A four-tone record consists of a duration integer followed by 12 long integers that contain the rate, phase, and pointers to the waveform descriptions of the four sounds (see Figure 7).

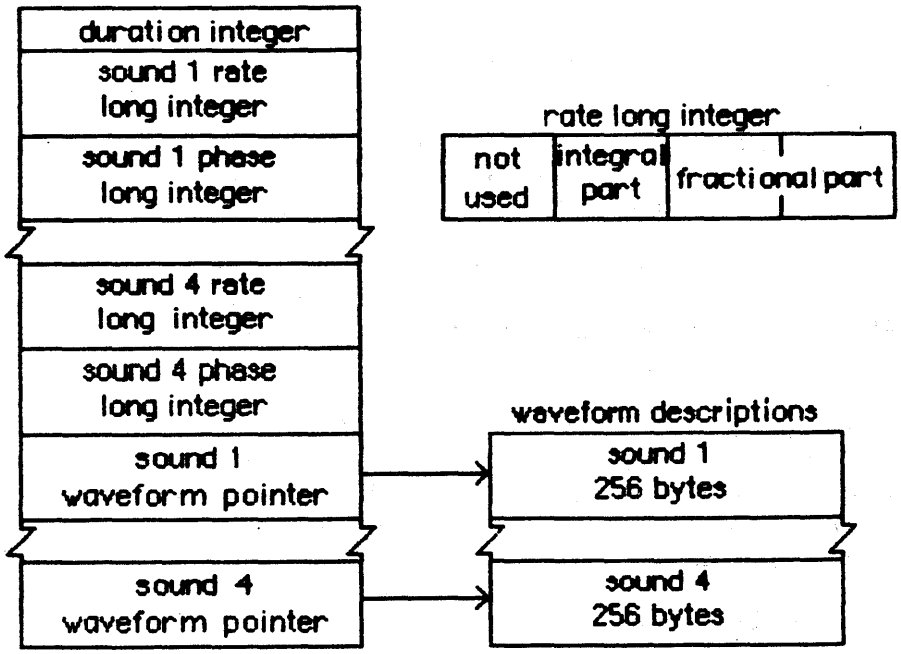


Figure 7. Four-Tone Record

The duration integer indicates the number of ticks that each sound will be generated, from Ø to 255 (Ø to 4.25 seconds). Each phase integer indicates the byte within the waveform description at which the synthesizer should begin producing sound (the first byte is byte number Ø). Each rate long integer determines the speed at which the synthesizer cycles through the waveform. The low-order word of the rate long integer contains the fractional part of the rate, and the low-order byte of the high-order word contains the integral part. (Binary fractions are described in the Toolbox Utilities manual under Fixed-Point Numbers.) The rate long integer can vary between Ø and 16777215.

The waveform description for each tone must contain 256 bytes. The four-tone synthesizer creates sound by starting at the byte in the waveform description specified by the phase, and skipping rate bytes ahead every 44.93 usec; when the time specified by the duration integer has elapsed, the synthesizer stops. The amount of time required to cycle completely through the waveform is 16777216 * 44.93 usec / rate (115Ø2 usec if the rate long integer is 65536--corresponding to about 87 Hz if the waveform contains one wavelength). If the waveform contains one wavelength, the frequency the rate corresponds to is given by

frequency (Hz) = rate / 753.795

The maximum rate of 16777215 corresponds to 44.93 usec, or about 22.3 kHz if the waveform contains one wavelength, and a rate of Ø produces no sound. A partial list of rate values and corresponding frequencies for notes comprising Ptolemy's diatonic scale (the scale to which pianos are tuned) isis given in the summary at the end of this manual.

USING THE SOUND DRIVER

The Sound Driver is a standard Macintosh device driver, and is manipulated via the Device Manager DriverOpen, DriverClose, Write, and Control calls. The Sound Driver doesn't support Read or Status calls.

The Sound Driver is automatically opened when the system starts up. Its driver name is .Sound, and its driver reference number is -4. To close the Sound Driver, call DriverClose(-4); you can reopen it by calling DriverOpen(".Sound").

To use one of the three types of synthesizers to generate sound, use the Memory Manager routines NewHandle and SetHandleSize to allocate heap space for a synthesizer buffer. Then, fill the buffer with values describing the sound, and make an Write call to the Sound Driver. The Write parameters passed must be as follows:

- RefNum must be -4.

- BuffPtr must point to the synthesizer buffer.

- Count must contain the length of the synthesizer buffer, in bytes.

When you use the free-form synthesizer, the amplitudes described by each byte in the waveform description are generated sequentially until the number of bytes specified by the count parameter have been written. When you use the square-wave synthesizer, the sounds described by each sound triplet are generated sequentially until either the end of the buffer has been reached (indicated by a count integer of Ø in the square-wave buffer), or the number of bytes specified by the Write call's count parameter have been written. When you use the four-tone synthesizer, all four sounds are generated for the length of time specified by the duration integer in the four-tone record.

There are three different calls you can make to the Sound Driver's control routine:

- KillIO is a standard control call supported by all drivers. It stops any sound currently being generated, and deletes all asynchronous I/O requests to the Sound Driver that haven't yet been processed.

- SetVolume allows you to change the volume of the sound that passes through the Macintosh speaker. There are eight levels of volume, specified by the three low-order bits in the opParam parameter of the Control call, Ø being low, and 7 high. Applications shouldn't change the speaker volume, as it's really up to the user to choose the normal sound level via the Control Panel desk accessory.

- Advanced Programmers: SetLevel enables you to control the amplitude of the sound generated by the square-wave synthesizer. The amplitude is contained in the opParam parameter of the Control call, and must be in the range Ø to 255. This call is explained in more detail below.

When you call the Sound Driver's control routine, the parameters must contain the following:

- RefNum must be -4.

- OpCode must specify the type of call:

Call	OpCode
KillIO	1
SetVolume	2
SetLevel	3

- OpParam must provide the volume level for a SetVolume call, and the amplitude for a SetLevel call.

(note)

Advanced programmers using low-level Pascal or assembly-language Device Manager routines must pass the above values in a parameter block. In addition, if you're calling the Sound Driver asynchronously, the

ioCompletion parameter must contain either the address of a completion routine or NIL.

Assembly-language note: The current speaker volume level is contained in the system global sdVolume.

Advanced Control Routine

The following paragraphs describe how the Sound Driver uses the Macintosh hardware to produce sound, and how you can intervene in the process to control the square-wave synthesizer. You can skip this section if it doesn't interest you, and you'll still be able to use the Sound Driver as described, except for the SetLevel call.

To generate sound at the amplitude level specified by a square-wave synthesizer buffer, the Sound Driver places the value of the amplitude integer into a 740-byte buffer shared by both the Sound Driver and the disk-motor speed-control circuitry. Then, every 44.93 usec when the video beam wraps from the right edge of the screen to the left, the microprocessor automatically fetches an additional two bytes from this buffer. The high-order byte is sent to the speaker, and the low-order byte to the disk-motor speed-control circuitry.

Assembly-language note: The amplitude level in the 740-byte buffer is contained in the system global soundLevel.

(note)

All the frequencies generated by the Sound Driver are multiples of this 44.93 usec period. The highest frequency the Sound Driver can physically generate corresponds to twice this period, 89.96 usec, or a frequency of 11116 Hz.

You can cause the square-wave synthesizer to start generating sound, and then change the amplitude of the sound being generated any time you wish:

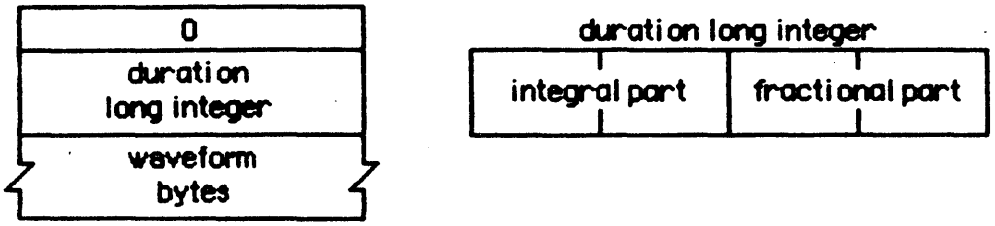
1. Make an asynchronous Write call to the Sound Driver specifying the count, amplitude, and duration of the sound you want generated. The amplitude you specify will be placed in the 740-byte buffer, and the Sound Driver will begin producing sound.

2. Whenever you want to change the sound being generated, make a SetLevel call with the opParam parameter specifying the amplitude of the new sound. The amplitude you specify will be placed in the 74Ø-byte buffer, and the sound will change. You can continue to change the sound until the time specified by the duration integer has elapsed.

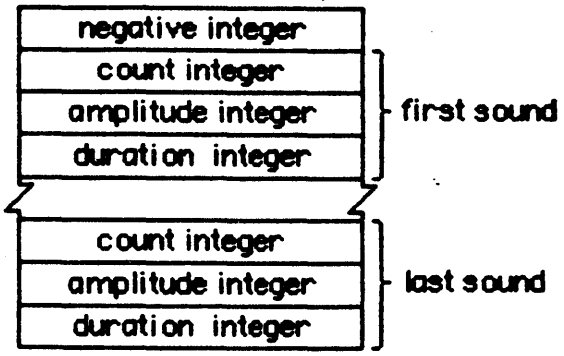
SUMMARY OF THE SOUND DRIVER

Data Structures

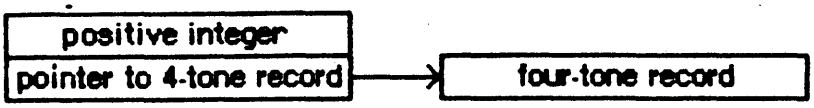
Free-Form Synthesizer Buffer



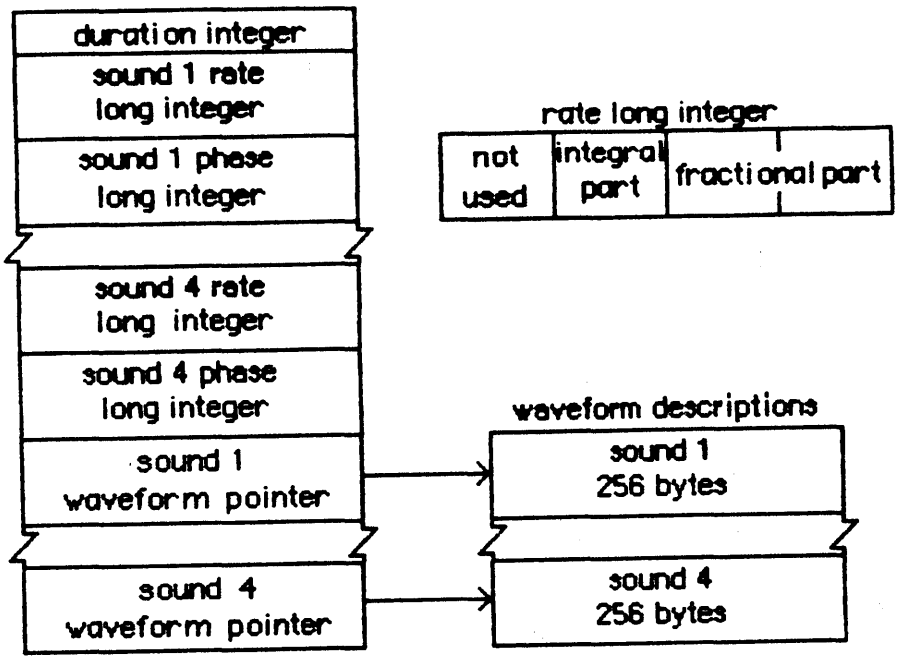
Square-Wave Synthesizer Buffer



Four-Tone Synthesizer Buffer



Four-Tone Record



Sound Driver Control Calls

Call	OpCode
KillIO	1
SetVolume	2
SetLevel	3

Assembly-Language Information

Variables

```
SdVolume            ;speaker volume level
SoundPtr            ;pointer to four-tone record
SoundBase           ;pointer to free-form buffer
SoundLevel          ;amplitude in 740-byte buffer
CurPitch            ;value of count in square-wave synthesizer buffer
```

Sound Driver Values For Notes Comprising Ptolemy's Diatonic Scale

Note (Freqency)	Rate Values for the Four-Tone Synthesizer		Count Values For the Square-Wave Synthesizer	
	Long Word	Long Integer	Word	Integer
3 octaves below middle C				
C (33)	ØØØØ 612B	24875	5CBA	23738
Db (35.2)	ØØØØ 67A5	26533	56EF	22255
D (37.125)	ØØØØ 6D5Ø	27984	526D	211Ø1
Eb (39.6)	ØØØØ 749A	2985Ø	4D46	19782
E (41.25)	ØØØØ 7976	31Ø94	4A2F	18991
F (44)	ØØØØ 818E	33166	458C	178Ø4
Gb (46.9375)	ØØØØ 8A35	35381	4131	16689
G (49.5)	ØØØØ 91CØ	37312	3DD1	15825
Ab (52.8)	ØØØØ 9B78	39800	39F4	14836
A (55)	ØØØØ A1F2	41458	37A3	14243
Bb (57.75)	ØØØØ AAØB	43531	34FD	13565
B (61.875)	ØØØØ B631	46641	3174	1266Ø
2 octaves below middle C				
C (66)	ØØØØ C256	4975Ø	2E5D	11869
Db (7Ø.4)	ØØØØ CF4B	53Ø67	2B77	11127
D (74.25)	ØØØØ DAA1	55969	2936	1Ø55Ø
Eb (79.2)	ØØØØ E934	597ØØ	26A3	9891
E (82.5)	ØØØØ F2EC	62188	2517	9495
F (88)	ØØØ1 Ø31D	66333	22C6	89Ø2
Gb (93.875)	ØØØ1 146A	7Ø762	2Ø99	8345
G (99)	ØØØ1 2381	74625	1EE9	7913
Ab (1Ø5.6)	ØØØ1 36FØ	796ØØ	1CFA	7418
A (11Ø)	ØØØ1 43E5	82917	1BD1	7121
Bb (115.5)	ØØØ1 5417	87Ø63	1A7E	6782
B (123.75)	ØØØ1 6C62	93282	18BA	633Ø
1 octave below middle C				
C (132)	ØØØ1 84AC	995ØØ	172F	5935
Db (14Ø.8)	ØØØ1 9E96	1Ø6134	15BC	5564
D (148.5)	ØØØ1 B542	111938	149B	5275
Eb (158.4)	ØØØ1 D269	1194Ø1	1351	4945
E (165)	ØØØ1 E5D8	124376	128C	4748
F (176)	ØØØ2 Ø63B	132667	1163	4451
Gb (187.75)	ØØØ2 28D5	141525	1Ø4C	4172
G (198)	ØØØ2 47Ø3	149251	ØF74	3956
Ab (211.2)	ØØØ2 6DE1	1592Ø1	ØE7D	37Ø9
A (22Ø)	ØØØ2 87CA	165834	ØDE9	3561
Bb (231)	ØØØ2 A82E	174126	ØD3F	3391
B (247.5)	ØØØ2 D8C4	186564	ØC5D	3165

Middle C					
C	(264)	ØØØ3 Ø959	199ØØ1	ØB97	2967
Db	(281.6)	ØØØ3 3D2C	212268	ØADE	2782
D	(297)	ØØØ3 6A85	223877	ØA4E	2638
Eb	(316.8)	ØØØ3 A4D2	2388Ø2	Ø9A9	2473
E	(33Ø)	ØØØ3 CBBØ	248752	Ø946	2374
F	(352)	ØØØ4 ØC77	265335	Ø8B1	2225
Gb	(375.5)	ØØØ4 51AA	283Ø5Ø	Ø826	2Ø86
G	(396)	ØØØ4 8EØ6	2985Ø2	Ø7BA	1978
Ab	(422.4)	ØØØ4 DBC3	3184Ø3	Ø73F	1855
A	(44Ø)	ØØØ5 ØF95	331669	Ø6F4	178Ø
Bb	(462)	ØØØ5 5Ø5D	348253	Ø6AØ	1696
B	(495)	ØØØ5 B188	373128	Ø62F	1583
1 octave above middle C					
C	(528)	ØØØ6 12B3	398ØØ3	Ø5CC	1484
Db	(563.2)	ØØØ6 7A59	424537	Ø56F	1391
D	(594)	ØØØ6 D5ØA	447754	Ø527	1319
Eb	(633.6)	ØØØ7 49A4	4776Ø4	Ø4D4	1236
E	(66Ø)	ØØØ7 976Ø	4975Ø4	Ø4A3	1187
F	(7Ø4)	ØØØ8 18EF	53Ø671	Ø459	1113
Gb	(751)	ØØØ8 A354	566ØØ	Ø413	1Ø43
G	(792)	ØØØ9 1CØD	597ØØ5	Ø3DD	989
Ab	(844.8)	ØØØ9 B786	6368Ø6	Ø39F	927
A	(88Ø)	ØØØA 1F2B	663339	Ø37A	89Ø
Bb	(924)	ØØØA AØBA	6965Ø6	Ø35Ø	848
B	(99Ø)	ØØØB 6311	746257	Ø317	791
2 octaves above middle C					
C	(1Ø56)	ØØØC 2567	796ØØ7	Ø2E6	742
Db	(1126.4)	ØØØC F4B2	849Ø74	Ø2B7	695
D	(1188)	ØØØD AA14	8955Ø8	Ø293	659
Eb	(1267.2)	ØØØE 9349	9552Ø9	Ø26A	618
E	(132Ø)	ØØØF 2EC1	995ØØ9	Ø251	593
F	(14Ø8)	ØØ1Ø 31DF	1Ø6134Ø	Ø22C	556
Gb	(15Ø2)	ØØ11 46A8	1132ØØ	Ø2ØA	522
G	(1584)	ØØ12 381B	1194Ø1Ø	Ø1EF	495
Ab	(1689.6)	ØØ13 6FØC	127361Ø	Ø1DØ	464
A	(176Ø)	ØØ14 3E57	1326681Ø	Ø1BD	445
Bb	(1848)	ØØ15 4175	1393Ø1Ø	Ø1A8	424
B	(198Ø)	ØØ16 C622	1492511Ø	Ø18C	396

3 octaves above middle C

C	(2112)	ØØ18 4ACF	1592Ø2Ø	Ø173	371
Db	(2252.8)	ØØ19 E965	169815Ø	Ø15C	348
D	(2376)	ØØ1B 5429	1791Ø2Ø	Ø14A	33Ø
Eb	(2534.4)	ØØ1D 2692	191Ø42Ø	Ø135	3Ø9
E	(264Ø)	ØØ1E 5D83	199ØØ2Ø	Ø129	297
F	(2816)	ØØ2Ø 63BF	212269Ø	Ø116	278
Gb	(3ØØ4)	ØØ22 8D5Ø	2264ØØ	Ø1Ø5	261
G	(3168)	ØØ24 7Ø36	2388Ø2Ø	ØØF7	247
Ab	(3379.2)	ØØ26 DE18	254722Ø	ØØE8	232
A	(352Ø)	ØØ28 7CAE	2653369	ØØDF	223
Bb	(3696)	ØØ2A 82EA	2786Ø3Ø	ØØD4	212
B	(396Ø)	ØØ2D 8C44	2985Ø3Ø	ØØC6	198

GLOSSARY

amplitude: The vertical distance between any given point on a wave and the horizontal line about which the amplitude oscillates.

four-tone record: A data structure describing the four tones produced by a four-tone synthesizer.

four-tone synthesizer: The part of the Sound Driver used to make simple harmonic tones, with up to four "voices" producing sound simultaneously.

free-form synthesizer: The part of the Sound Driver used to make complex music and speech.

frequency: The number of cycles per second (also called Hertz) at which a wave oscillates.

period: The time elapsed during one complete cycle of a wave.

phase: Some fraction of a wave cycle (measured from a fixed point on the wave).

square-wave synthesizer: The part of the Sound Driver used to produce less harmonic sounds such as beeps.

synthesizer buffer: A description of the sound to be generated by a synthesizer.

waveform: The physical shape of a wave.

waveform description: A sequence of bytes describing a waveform.

wavelength: The horizontal extent of one complete cycle of a wave.

MACINTOSH USER EDUCATION

Macintosh Serial Communication: A Programmer's Guide /DRIVER/SERIAL

See Also: The Macintosh User Interface Guidelines
Macintosh Operating System Manual
The Device Manager: A Programmer's Guide
Programming Macintosh Applications in Assembly Language

Modification History: First Draft (ROM 7) Bradley Hacker 5/13/84

*** Preliminary Draft. Not for distribution *** ABSTRACT

The Macintosh RAM Serial Driver and ROM Serial Driver are sets of data types and routines in the Macintosh Operating System for handling asynchronous serial communication between a Macintosh application and a serial device. This manual describes the Serial Drivers in detail.

TABLE OF CONTENTS

ABOUT THIS MANUAL

The Macintosh RAM Serial Driver and ROM Serial Driver are sets of data types and routines in the Macintosh Operating System for handling asynchronous serial communication between a Macintosh application and serial devices. This manual describes the Serial Drivers in detail. *** Eventually it will become part of the comprehensive Inside Macintosh manual. ***

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal. You should also be familiar with the following:

- the basic concepts behind the Macintosh Operating System's Memory Manager
- interrupts and the use of devices and device drivers, as described in the Device Manager Manual
- asynchronous serial data communication

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly-language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it.

The manual begins with an introduction to the Serial Drivers and what you can do with them. It then describes how you can use Serial Driver calls in your application to communicate with serial devices. This includes a detailed description of the Serial Drivers' routines--their parameters, calling protocol, effects, and so on.

Finally, there's a summary of the Serial Driver data structures and routine calls, for quick reference, followed by a glossary of terms used in this manual.

SERIAL COMMUNICATION

There are two Macintosh device drivers for serial communication: the RAM Serial Driver and the ROM Serial Driver. The two drivers are nearly identical, although the RAM driver has a few features the ROM driver doesn't. Both allow Macintosh applications to communicate with serial devices via the two RS-232/RS-422 serial ports on the back of the Macintosh.

The Serial Drivers support full-duplex asynchronous serial communication. Serial data is transmitted over a single-path communication line, one bit at a time (as opposed to parallel data, which is transmitted over a multiple-path communication line, multiple bits at a time). Full-duplex means that the Macintosh and another serial device connected to it can transmit data simultaneously (as

opposed to half-duplex operation, in which data can only be transmitted by one device at a time). Asynchronous communication means that the Macintosh and other serial devices communicating with it don't share a common timer, and no timing data is transmitted. The time interval between characters transmitted asynchronously can be of any length. The format of asynchronous serial data communication used by the Serial Drivers is shown in Figure 1.

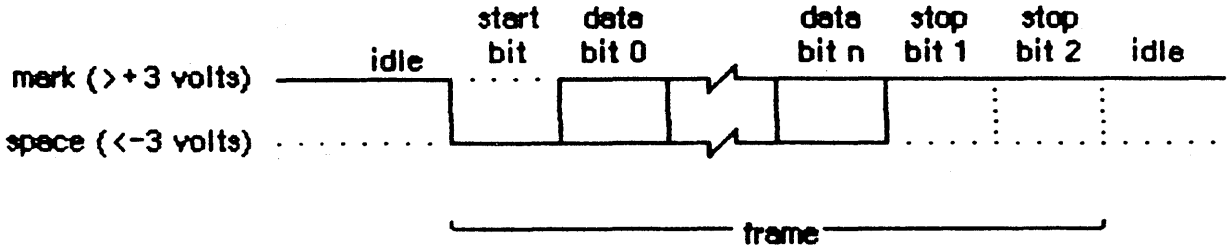


Figure 1. Asynchronous Data Transmission

When a transmitting serial device is idle (not sending data), it maintains the transmission line in a continuous state ("mark" in Figure 1). The transmitting device may begin sending a character at any time by sending a start bit. The start bit tells the receiving device to prepare to receive a character. The transmitting device then transmits 5, 6, 7, or 8 data bits, optionally followed by an even or odd parity bit. If a parity bit is transmitted, its value is chosen such that the number of 1's among the data bits and the parity bit is even or odd, depending on whether the parity is even or odd, respectively. Finally, the transmitting device sends 1, 1.5, or 2 stop bits, indicating the end of the character.

If a parity bit is set incorrectly, the receiving device will note a parity error. The time elapsed from the start bit to the last stop bits is called a frame. If the receiving device doesn't get a stop bit after the data and parity bits, it will note a framing error. After the stop bits, the transmitting device may send another character or maintain the line in the mark state. If the line is held in the "space" state (Figure 1) for one frame or longer, a break occurs. Breaks are used to interrupt data transmission.

ABOUT THE SERIAL DRIVERS

Each Serial Driver actually consists of four drivers: one input driver and one output driver for the modem port, and one input driver and one output driver for the printer port (Figure 2). Each input driver receives data via a serial port and transfers it to the application. Each output driver takes data from the application and sends it out through a serial port. The input and output drivers for a port are closely related, and share some of the same routines. Each driver does, however, have a separate device control entry, which allows the Serial Drivers to support full-duplex communication. An individual

port can both transmit and receive data at the same time. The serial ports are controlled by the Macintosh's Zilog Z853Ø Serial Communications Controller (SCC). Channel A of the SCC controls the modem port, and channel B controls the printer port.

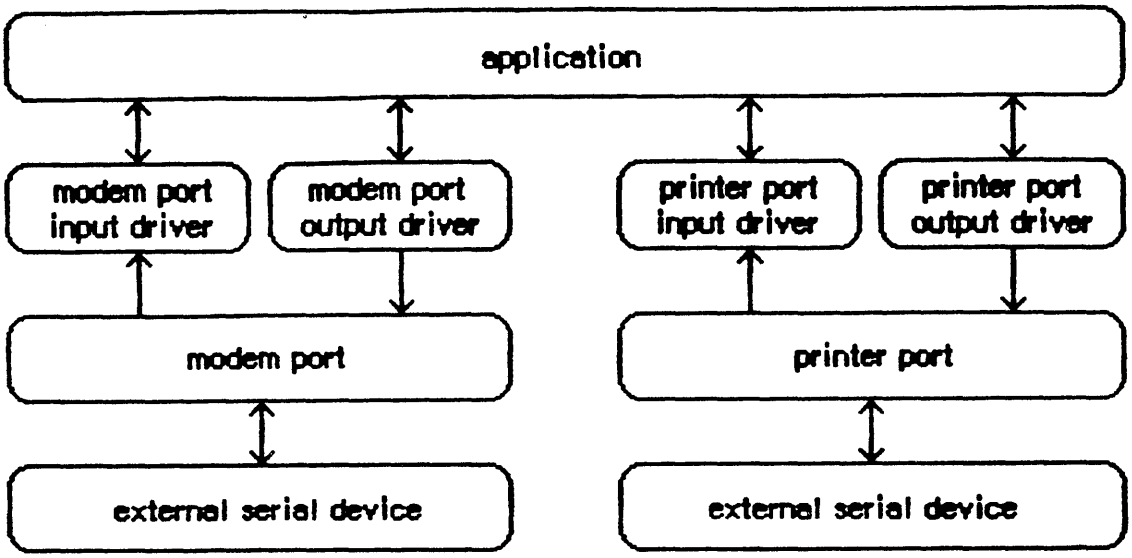


Figure 2. Input and Output Drivers of a Serial Driver

Data received via a serial port passes through a 3-character buffer in the SCC and then into a buffer in the input driver for the port. Characters are removed from the input driver's buffer each time an application issues a Read call to the driver. Each input driver's buffer can initially hold up to 64 characters, but your application can increase this if necessary. If the SCC buffer ever becomes full, a hardware overrun error occurs. If an input driver's buffer ever becomes full, a software overrun error occurs.

The printer port should be used for output-only connections to devices such as printers, or at low baud rates (3ØØ baud or less). The modem port has no such restrictions. It may be used simultaneously with disk accesses without fear of hardware overrun errors, because whenever the Disk Driver must turn off interrupts for longer than 1ØØ usec, it stores any data received via the modem port and later passes the data to the modem port's input driver.

All four drivers default to 96ØØ baud, eight data bits per character, no parity bit, and two stop bits. You can change any of these options. The Serial Drivers supports CTS hardware handshaking and XOn/XOff software flow control. Each driver defaults to hardware handshake only, your application must enable XOn/XOff flow control if needed.

(note)

The ROM Serial Driver doesn't support XOn/XOff input flow control--only output flow control. Use the RAM Serial Driver if you want XOn/XOff input flow control.

Whenever an input driver receives a break, it terminates any pending Read requests, but not Write requests. You can choose to have the input drivers terminate Read requests whenever a parity, overrun, or framing error occurs.

(note)

The ROM Serial Driver always terminates input requests when an error occurs. Use the RAM Serial Driver if you don't want input requests to be terminated by errors.

You can request the Serial Drivers to post device driver events whenever a change in the hardware handshake status or a break occurs, if you want your application to take some specific action upon these occurrences.

USING THE SERIAL DRIVERS

You can call the Serial Drivers via high-level Pascal, low-level Pascal, and assembly-language Device Manager routines. The information in this section is oriented toward the high-level Pascal calls, and advanced programmers will need to consult the Device Manager manual for equivalent low-level Pascal or assembly-language routines.

Drivers are referred to by name and reference number as shown below:

Driver	Driver name	Reference number
Modem port input	.AIn	-6
Modem port output	.AOut	-7
Printer port input	.BIn	-8
Printer port output	.BOut	-9

Before you can send or receive data through a port, both the input and output drivers for the port must be opened. ~~All four ROM drivers are automatically opened when the system starts up.~~ To open the RAM Serial Driver, call RAMSDOpen.

You must open ROM SDs yourself.

When you open an output driver, the Serial Driver initializes local variables for the output driver and the associated input driver, allocates and locks buffer storage for both drivers, installs interrupt vectors for both drivers, and initializes the correct SCC channel for the output driver only. When you open an input driver, the Serial Driver only notes the location of its device control entry. You must open both the input and output drivers for a port before you can use it, but the order in which the drivers are opened doesn't matter.

If you would like to reclaim the space occupied by a driver's storage, you can can call DriverClose to close the ROM Serial Driver, and RAMSDClose to close the RAM Serial Driver. When you close an output driver, the Serial Driver resets the appropriate SCC channel, configures the driver for external/status (mouse) interrupts only, releases all local variable and buffer storage space, and restores any changed interrupt vectors. Closing an input driver has no effect. The

ROM Serial Driver is automatically closed when you call RAMSDOpen, and opened when you call RAMSDClose.

(warning)
You should not close the ROM Serial Driver unless you're immediately going to open a RAM Serial Driver for the same port; otherwise mouse interrupts will be lost.

To transmit serial data out through a port, make a Device Manager Write call to the output driver for the port. You must pass the following parameters:

- RefNum must be -7 or -9, depending on whether you're using the modem port or printer port.
- BuffPtr must point to the data you want to transmit.
- Count must contain the number of bytes you want to transmit.

To receive serial data from a port, make a Device Manager Read call to the input driver for the port. You must pass the following parameters:

- RefNum must be -6 or -8, depending on whether you're using the modem port or printer port.
- BuffPtr must point to the location where you want to receive the data.
- Count must contain the number of bytes you want to receive.

There are six different calls you can make to the Serial Driver's control routine:

- KillIO causes all current I/O requests to be aborted and any bytes remaining in both input buffers to be discarded. KillIO is a Device Manager call.
- SerReset resets and reinitializes a driver.
- SerSetBuf allows you to specify a new input buffer.
- SerHShake allows you to specify handshake options.
- SerSetBrk sets break mode.
- SerClrBrk clears break mode.

There are two different calls you can make to the Serial Driver's status routine:

- SerGetBuf returns the number of available buffered bytes.
- SerErrFlag returns information about errors, I/O requests, and handshake.

SERIAL DRIVER ROUTINES

This section describes the calls that you can make to the Serial Driver's control and status routines. The routine names given here can be used only from Pascal; assembly-language programmers must make equivalent Control and Status calls.

FUNCTION RAMSDOpen : OSErr; *** Not yet implemented ***

RAMSDOpen closes the ROM Serial Driver and opens the RAM Serial Driver.

FUNCTION RAMSDClose : OSErr; *** Not yet implemented ***

RAMSDClose closes the RAM Serial Driver and opens the ROM Serial Driver.

FUNCTION SerReset(refNum: INTEGER; serConfig: INTEGER) : OSErr;

SerReset resets and reinitializes the driver having the reference number refNum according to the information in serConfig. Figure 3 shows the format of serConfig.

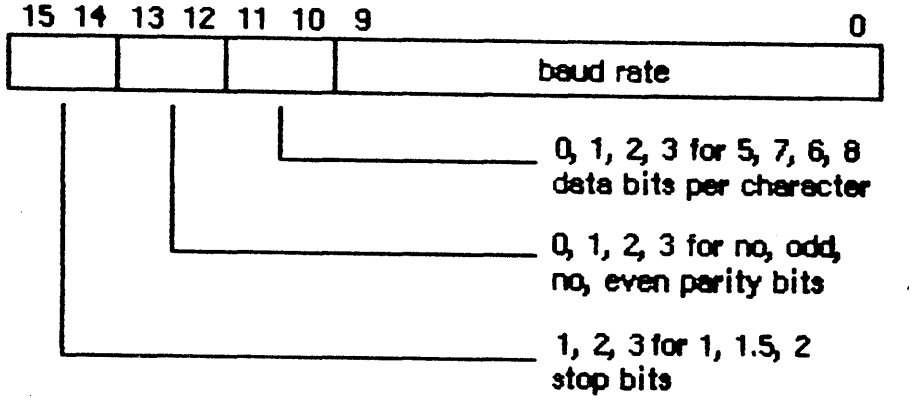


Figure 3. Driver Reset Information

Use can use the following predefined constants to test or set the value of various bits of serConfig:

```
CONST baud300    =    380; {300 baud}
      baud600    =    189; {600 baud}
      baud1200   =     94; {1200 baud}
      baud1800   =     62; {1800 baud}
      baud2400   =     46; {2400 baud}
      baud3600   =     30; {3600 baud}
      baud4800   =     22; {4800 baud}
      baud7200   =     14; {7200 baud}
      baud9600   =     10; {9600 baud}
      baud19200  =      4; {19200 baud}
      baud57600  =      0; {57600 baud}
      stop10     =  16384; {set for 1 stop bit}
      stop15     = -32768; {set for 1.5 stop bits}
      stop20     = -16384; {set for 2 stop bits}
      noParity   =   8192; {set for no parity}
      oddParity  =   4096; {set for odd parity}
      evenParity =  12288; {set for even parity}
      data5      =      0; {set for 5 data bits}
      data6      =   2048; {set for 6 data bits}
      data7      =   1024; {set for 7 data bits}
      data8      =   3072; {set for 8 data bits}
```

For example, the default setting of 9600 baud, eight data bits, two stop bits, and no parity bit is equivalent to baud9600+data8+stop20+noParity.

Assembly-language note: SerReset is equivalent to a Control call with csCode = 8.

```
FUNCTION SerSetBuf(refNum: INTEGER; serBPtr: Ptr; serBLen: INTEGER) :
        OSErr;
```

SerSetBuf specifies a new input buffer for the driver having the reference number refNum. SerBPtr points to the buffer, and serBLen specifies the number of bytes in the buffer minus 2.

(warning)
You must lock this buffer while it's in use.

Assembly-language note: SerSetBuf is equivalent to a Control call with csCode = 9.

```
FUNCTION SerHShake(refNum: INTEGER; flags: SerShk) : OSErr;
```

SerHShake sets handshake options and other control information for the driver having the reference number refNum. The flags parameter is a record with the following data structure:

```
TYPE SerShk = PACKED RECORD
                fXOn: Byte; {XOn/XOff output flow control enabled}
                fCTS: Byte; {CTS hardware handshake enabled}
                xOn:  CHAR; {XOn character}
                xOff: CHAR; {XOff character}
                errs: Byte; {errors that cause abort}
                evts: Byte; {status changes that are events}
                fInX: Byte; {XOn/XOff input flow control enabled}
                null: Byte  {not used}
              END;
```

If fXOn is nonzero, XOn/XOff output flow control is enabled. XOn and xOff specify the XOn character and XOff character used for XOn/XOff flow control. If fInX is nonzero, XOn/XOff input flow control is enabled. If fCTS is nonzero, CTS hardware handshake is enabled. The errs field indicates which errors will cause abort of input requests; you can use the following predefined constants to set or test the value of fCTS:

```
CONST parityErr    = 16; {set if parity error will cause abort}
      hwOverrunErr = 32; {set if hardware overrun error will }
                         { cause abort}
      framingErr   = 64; {set if framing error will cause abort}
```

(note)
The ROM Serial Driver doesn't support XOn/XOff input flow control or aborts caused by error conditions.

The evts field indicates whether changes in the CTS or break status will cause the Serial Driver to post device driver events; you can use the following predefined constants to set or test the value of evts:

```
CONST ctsEvent   =  32; {set if CTS change will cause event to }
                        { be posted}
      breakEvent = 128; {set if break status change will cause }
                        { event to be posted}}
```

(warning)
Use of this option is discouraged because of the long time that interrupts are disabled while such an event is posted.

Assembly-language note: SerHShake is equivalent to a Control call with csCode = 10.

FUNCTION SerSetBrk(refNum: INTEGER) : OSErr;

SerSetBrk sets break mode in the driver having the reference number refNum.

Assembly-language note: SerSetBrk is equivalent to a Control call with csCode = 12.

FUNCTION SerClrBrk(refNum: INTEGER) : OSErr;

SerClrBrk clears break mode in the driver having the reference number refNum, by reinitializing the appropriate SCC channel.

Assembly-language note: SerClrBrk is equivalent to a Control call with csCode = 11.

FUNCTION SerGetBuf(refNum: INTEGER; VAR count: LongInt) : OSErr;

SerGetBuf returns the number of unread bytes, in count, in buffer of the input driver having the reference number refNum.

Assembly-language note: SerGetBuf is equivalent to a Status call with csCode = 2.

FUNCTION SerStatus(refNum: INTEGER; serSta: SerStaRec) : OSErr;

SerErrFlag returns three words of status information for the driver having the reference number refNum. The serSta parameter is a record with the following data structure:

```
TYPE SerStaRec = PACKED RECORD
                   cumErrs:  Byte; {cumulative errors}
                   xOffSent: Byte; {XOff sent flag}
                   rdPend:   Byte; {read pending flag}
                   wrPend:   Byte; {write pending flag}
                   ctsHold:  Byte; {CTS flow control hold flag}
                   xOffHold: Byte  {XOff flow control hold flag}
                 END;
```

CumErrs indicates which errors have occurred since the last time SerStatus was called:

```
CONST swOverrunErr =  1; {set if software overrun error occurred}
      parityErr    = 16; {set if parity error occurred}
      hwOverrunErr = 32; {set if hardware overrun error occurred}
      framingErr   = 64; {set if framing error occurred}
```

If xOffSent equals 128Ø, then the driver has sent an XOff character. If rdPend is nonzero, the driver has a Read call pending. If wrPend is nonzero, the driver has a Write call pending. If ctsHold is nonzero, then output has been suspended because the hardware handshake was negated. If xOffHold is nonzero, then output has been suspended because an XOff character was received.

Assembly-language note: SerStatus is equivalent to a Status call with csCode = 8.

SUMMARY OF THE SERIAL DRIVERS

Constants

```
CONST { Driver reset information }

      baud300    =    380; {300 baud}
      baud600    =    189; {600 baud}
      baud1200   =     94; {1200 baud}
      baud1800   =     62; {1800 baud}
      baud2400   =     46; {2400 baud}
      baud3600   =     30; {3600 baud}
      baud4800   =     22; {4800 baud}
      baud7200   =     14; {7200 baud}
      baud9600   =     10; {9600 baud}
      baud19200  =      4; {19200 baud}
      baud57600  =      0; {57600 baud}
      stop10     =  16384; {set for 1 stop bit}
      stop15     = -32768; {set for 1.5 stop bits}
      stop20     = -16384; {set for 2 stop bits}
      noParity   =   8192; {set for no parity}
      oddParity  =   4096; {set for odd parity}
      evenParity =  12288; {set for even parity}
      data5      =      0; {set for 5 data bits}
      data6      =   2048; {set for 6 data bits}
      data7      =   1024; {set for 7 data bits}
      data8      =   3072; {set for 8 data bits}

      { Errors }

      swOverrunErr =  1; {set if software overrun error}
      parityErr    = 16; {set if parity error}
      hwOverrunErr = 32; {set if hardware overrun error}
      framingErr   = 64; {set if framing error}

      { Changes that cause events to be posted }

      ctsEvent   =  32; {set if CTS change will cause event to }
                        { be posted}
      breakEvent = 128; {set if break status change will cause }
                        { event to be posted}
```

Data Structures

```
TYPE SerShk = PACKED RECORD
                fXOn: Byte; {XOn/XOff output flow control enabled}
                fCTS: Byte; {CTS hardware handshake enabled}
                xOn:  CHAR; {XOn character}
                xOff: CHAR; {XOff character}
                errs: Byte; {errors that cause abort}
```

```
            evts: Byte; {status changes that are events}
            fInX: Byte; {XOn/XOff input flow control enabled}
            null: Byte  {not used}
          END;

SerStaRec = PACKED RECORD
              cumErrs:  Byte; {cumulative errors}
              xOffSent: Byte; {XOff sent flag}
              rdPend:   Byte; {read pending flag}
              wrPend:   Byte; {write pending flag}
              ctsHold:  Byte; {CTS flow control hold flag}
              xOffHold: Byte  {XOff flow control hold flag}
            END;
```

Serial Driver Routines

Opening and Closing RAM Serial Drivers

```
FUNCTION RAMSDOpen  : OSErr;
FUNCTION RAMSDClose : OSErr;
```

Changing Serial Driver Information

```
FUNCTION SerReset (refNum: INTEGER; serConfig: INTEGER) : OSErr;
FUNCTION SerSetBuf(refNum: INTEGER; serBPtr: Ptr; serBLen: INTEGER) :
                   OSErr;
FUNCTION SerHShake(refNum: INTEGER; flags: SerShk) : OSErr;
FUNCTION SerSetBrk(refNum: INTEGER) : OSErr;
FUNCTION SerClrBrk(refNum: INTEGER) : OSErr;
```

Getting Serial Driver Information

```
FUNCTION SerGetBuf(refNum: INTEGER; VAR count: LongInt) : OSErr;
FUNCTION SerStatus(refNum: INTEGER; serSta: SerStaRec) : OSErr;
```

Assembly-Language Information

Structure of Control Information *not yet available***

fXOn	XOn/XOff output flow control enabled
fCTS	CTS hardware handshake enabled
xOn	XOn character
xOff	XOff character
errs	Errors that cause abort
evts	Status changes that are events
fInX	XOn/XOff input flow control enabled

Structure of Status Information *** nyet ***

cumErrs	Cumulative errors
xOffSent	XOff sent flag
rdPend	Read pending flag
wrPend	Write pending flag
ctsHold	CTS flow control hold flag
xOffHold	XOff flow control hold flag

GLOSSARY

asynchronous communication: A method of data transmission where the receiving and sending devices don't share a common timer, and no timing data is transmitted.

break: The condition resulting when a device maintains its transmission line in the space state for at least one frame.

data bits: Data communication bits that encode transmitted characters.

frame: The time elapsed from the start bit to the last stop bit.

framing error: The condition resulting when a device doesn't receive a stop bit when expected.

full-duplex communication: A method of data transmission where two devices transmit data simultaneously.

hardware overrun error: The condition that occurs when the SCC's buffer becomes full.

input driver: A device driver that receives serial data via a serial port and transfers it to an application.

mark state: The state of a transmission line indicating a binary '1'.

output driver: A device driver that receives data via a serial port and transfers it to an application.

overrun error: See hardware overrun error and software overrun error.

parity bit: A data communication bit used to verify that data bits received by a device match the data bits transmitted by another device.

parity error: The condition resulting when the parity bit received by a device isn't what was expected.

serial data: Data communicated over a single-path communication line, one bit at a time.

software overrun error: The condition that occurs when an input driver's buffer becomes full.

space state: The state of a transmission line indicating a binary 'Ø'.

start bit: A serial data communications bit that signals that the next bits transmitted are data bits.

stop bit: A serial data communications bit that signals that the end of data bits.

MACINTOSH USER EDUCATION

The Vertical Retrace Manager: A Programmer's Guide /VRMGR/TASK

See Also: The Macintosh User Interface Guidelines
The Memory Manager: A Programmer's Guide
The File Manager: A Programmer's Guide
The Device Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Desk Manager: A Programmer's Guide
Inside Macintosh: A Road Map
Programming Macintosh Applications in Assembly Language

Modification History: First Draft (ROM 7) Bradley Hacker 6/15/84

ABSTRACT

This manual describes the Vertical Retrace Manager, the part of the Macintosh Operating System that schedules and performs recurrent tasks during vertical retrace interrupts. It describes how your application can install and remove its own recurrent tasks.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Vertical Retrace Manager, the part of the Macintosh Operating System that schedules and performs recurrent tasks during vertical retrace interrupts. It describes how your application can install and remove its own recurrent tasks. *** Eventually it will become part of the comprehensive Inside Macintosh manual. ***

Like all Operating System documentation, this manual assumes you're familiar with Lisa Pascal. You should also be familiar with the following:

- the Macintosh Operating System's Memory Manager
- interrupts, as described in the Macintosh Operating System's Device Manager manual
- queues, as described in the Operating System Utilities manual *** not yet; for now, see the appendix of the File Manager manual. ***

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly-language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it.

The manual begins with an introduction to the Vertical Retrace Manager and what you can do with it. It then introduces the routines of the Vertical Retrace Manager and tells how they fit into the flow of your application. This is followed by detailed descriptions of the routines themselves.

Finally, there's a summary of the Vertical Retrace Manager, for quick reference, followed by a glossary of terms used in this manual.

ABOUT THE VERTICAL RETRACE MANAGER

The Macintosh video circuitry generates a vertical retrace interrupt (also known as the vertical blanking or VBL interrupt) 60 times a second while the beam of the display tube returns from the bottom of the screen to the top to display the next frame. The Operating System uses this interrupt as a convenient time to perform the following sequence of recurrent tasks:

1. Increment the number of ticks since system startup (every interrupt). (You can get this number by calling the Toolbox Event Manager function TickCount.)
2. Check whether the stack and heap have collided (every interrupt).

3. Handle cursor movement (every interrupt).

4. Post a mouse event if the state of the mouse button changed from its previous state and then remained unchanged for four interrupts (every other interrupt).

5. Post a disk inserted event if a disk has been inserted (every 3Ø interrupts).

These tasks must execute at regular intervals based on the "heartbeat" of the Macintosh, and shouldn't be changed.

An application can add any number of its own tasks for the Vertical Retrace Manager to execute. Application tasks can perform any desired actions as long as memory is neither allocated nor released, and can be set to execute at any frequency (up to once per vertical retrace interrupt). For example, a task within an electronic-mail application might check every tenth of a second to see if it has received any messages.

(note)

Application tasks longer than about one-sixtieth of a second will affect other interrupt-driven parts of the Macintosh, such as the mouse position.

Information describing each application task is contained in the vertical retrace queue. The vertical retrace queue is a standard Macintosh Operating System queue, as described in the Operating System Utilities manual *** doesn't yet exist; for now, see the File Manager manual's appendix ***. Each entry in the vertical retrace queue has the following structure:

```
TYPE VBLTask = RECORD
                 qLink:    QElemPtr; {next queue entry}
                 qType:    INTEGER;  {queue type}
                 vblAddr:  ProcPtr;  {task address}
                 vblCount: INTEGER;  {task frequency}
                 vblPhase: INTEGER   {task phase}
               END;
```

As in all Operating System queue entries, qLink points to the next entry in the queue, and qType indicates the queue type. QType should always be ORD(vType) in the vertical retrace queue.

VBLAddr contains the address of the task. VBLCount specifies the number of ticks between successive calls to the task. This value is decremented each sixtieth of a second until it reaches Ø, at which point the task is called. The task must then reset vblCount, or its entry will be removed from the queue after it has been executed. VBLPhase contains an integer (smaller than vblCount) used to modify vblCount when the task is first added to the queue. This ensures that two or more routines added to the queue at the same time with the same vblCount value will be out of phase with each other, and won't be called during the same interrupt.

Assembly-language note: The Vertical Retrace Manager sets bit 6 of the queue flags whenever a task is being executed; assembly-programmers can use the global constant inVBL to test this bit.

USING THE VERTICAL RETRACE MANAGER

This section discusses how the Vertical Retrace Manager routines fit into the general flow of an application program. The routines themselves are described in detail in the next section.

The Vertical Retrace Manager is automatically initialized each time the system is started up. To add an application task to the vertical retrace queue, call VInstall. When your application no longer wants a task to be executed, it can remove the task from the vertical retrace queue by calling VRemove. An application task shouldn't call VRemove to remove its entry from the queue--either the application should call VRemove, or the task should simply not reset the vblCount field of the queue entry.

An application task cannot call routines that cause memory to be allocated or released. This severely limits the actions of tasks, so you might prefer using the Desk Manager procedure SystemTask to perform periodic actions. Or, since the very first thing the Vertical Retrace Manager does during a vertical retrace interrupt is increment the tick count, your application could call the Toolbox Event Manager function TickCount repeatedly and perform periodic actions whenever a specific number of ticks have elapsed.

Assembly-language note: Application tasks may use registers DØ through D3 and AØ through A3, and must save and restore any additional registers used. They must exit with an RTS instruction.

If you'd like to manipulate the contents of the vertical retrace queue directly, you can get a pointer to the vertical retrace queue by calling GetVBLQHdr.

VERTICAL RETRACE MANAGER ROUTINES

This section describes the Vertical Retrace Manager routines. Each routine is presented in its Pascal form; where applicable, it's followed by a box containing information needed to use the routine from assembly language. For general information on using the Vertical Retrace Manager from assembly language, see the manual Programming Macintosh Applications in Assembly Language.

```
FUNCTION VInstall (vblTaskPtr: QElemPtr) : OSErr;
```

Trap macro	_VInstall	
On entry	AØ:	vblTaskPtr (pointer)
On exit	DØ:	result code (integer)

VInstall adds the task described by vblTaskPtr to the vertical retrace queue. Your application must fill in all fields of the task except qLink. VInstall returns one of the result codes listed below.

Result codes	noErr	No error
	vTypErr	QType field isn't ORD(vType)

```
FUNCTION VRemove (vblTaskPtr: QElemPtr) : OSErr;
```

Trap macro	_VRemove	
On entry	AØ:	vblTaskPtr (pointer)
On exit	DØ:	result code (integer)

VRemove removes the task described by vblTaskPtr from the vertical retrace queue. It returns one of the result codes listed below.

Result codes	noErr	No error
	vTypErr	QType field isn't ORD(vType)
	qErr	Task entry isn't in the queue

FUNCTION GetVBLQHdr : QHdrPtr; [Pascal only]

GetVBLQHdr returns a pointer to the vertical retrace queue.

Assembly-language note: To access the contents of the vertical retrace queue from assembly language, assembly-language programmers can use offsets from the address of the global variable vblQueue.

SUMMARY OF THE VERTICAL RETRACE MANAGER

Constants

```
CONST { Result codes }

      noErr    =  Ø; {no error}
      qErr     = -1; {task entry isn't in the queue}
      vTypErr  = -2; {qType field isn't ORD(vType)}
```

Data Types

```
TYPE VBLTask = RECORD
                 qLink:    QElemPtr; {next queue entry}
                 qType:    INTEGER;  {queue type}
                 vblAddr:  ProcPtr;  {task address}
                 vblCount: INTEGER;  {task frequency}
                 vblPhase: INTEGER   {task phase}
               END;
```

Routines

```
FUNCTION VInstall (vblTaskPtr: QElemPtr) : OSErr;
FUNCTION VRemove  (vblTaskPtr: QElemPtr) : OSErr;
FUNCTION GetVBLQHdr : QHdrPtr;  [Pascal only]
```

Assembly-Language Information

Constants

```
inVBL        .EQU      6        ;set if Vertical Retrace Manager
                                ; is executing

; Result codes

qErr         .EQU      -1       ;task entry isn't in the queue
vTypErr      .EQU      -2       ;qType field isn't vType
```

Vertical Retrace Queue Entry

```
qLink              Pointer to next queue entry
qType              Queue type
vblAddr            Task address
vblCount           Task frequency
vblPhase           Task phase
```

Variables

Name	Size	Contents
vblQueue	4 bytes	Vertical retrace queue

GLOSSARY

vertical retrace interrupt: The interrupt that occurs 6Ø times a second while the beam of the display tube returns from the bottom of the screen to the top to display the next frame.

vertical retrace queue: A list of the application tasks to be executed during the vertical retrace interrupt.

COMMENTS?

Macintosh User Education encourages your comments on this manual.

- What do you like or dislike about it?

- Were you able to find the information you needed?

- Was it complete and accurate?

- Do you have any suggestions for improvement?

Please send your comments to the author (indicated on the cover page) at 10460 Bandley Drive M/S 3-G, Cupertino CA 95014. Mark up a copy of the manual or note your remarks separately. (We'll return your marked-up copy if you like.)

Thanks for your help!

Macintosh System Errors

System Trouble (bomb) Alert ID definitions

Assembly Label	Error #	Description
DSSysErr	32767	general system error
DSBusError	1	bus error
DSAddressErr	2	address error
DSIllInstErr	3	illegal instruction error
DSZeroDivErr	4	zero divide error
DSChkErr	5	check trap error
DSOvFlowErr	6	overflow trap error
DSPrivErr	7	privelege violation error
DSTraceErr	8	trace mode error
DSLineAErr	9	line 1010 trap error
DSLineFErr	10	line 1111 trap error
DSMiscErr	11	miscellaneous hardware exception error
DSCoreErr	12	unimplemented core routine error
DSIrqErr	13	uninstalled interrupt error
DSIOCoreErr	14	IO Core Error
DSLoadErr	15	Segment Loader Error
DSFPErr	16	Floating point error
DSNoPackErr	17	package 0 not present
DSNoPk1	18	package 1 not present
DSNoPk2	19	package 2 not present
DSNoPk3	20	package 3 not present
DSNoPk4	21	package 4 not present
DSNoPk5	22	package 5 not present
DSNoPk6	23	package 6 not present
DSNoPk7	24	package 7 not present
DSMemFullErr	25	out of memory!
DSBadLaunch	26	can't launch file
DSFSErr	27	file system map has been trashed
DSStknHeap	28	stack has moved into application heap
DSReInsert	30	request user to reinsert off-line volume
DSNotThe1	31	not the disk I wanted

Memory Manager trouble codes (System Trouble IDs)

mtSetLog	32	Set Logical Size Error
mtAdjFre	33	Adjust Free Error
mtAdjCnt	34	Adjust Counters Error
mtMkeBkf	35	Make Block Free Error
mtSetSiz	36	Set Size Error
mtInitMem	37	Initialize Memory Manager Error
mtBCerr	38	
mtCZerr	39	
mtCZ1err	40	
mtCZ2err	41	
mtCZ3err	42	
mtEqCerr	43	
mtEvCerr	44	
mtHCerr	45	
mtPCerr	46	
mtSCerr	47	
mtRC1err	48	
mtRC2err	49	
mtSABerr	50	
mtACerr	51	
mtIZCerr	52	
mtPrCerr	53	

General System Errors (VBL Mgr, Queueing, Etc.)

Assembly Label	Error #	Description
NoErr	0	success is absence of errors
QErr	-1	queue element not found during deletion
VTypErr	-2	invalid queue element
CorErr	-3	core routine number out of range
UnimpErr	-4	unimplemented core routine

I/O System Errors

ControlErr	-17	
StatusErr	-18	
ReadErr	-19	
WritErr	-20	
BadUnitErr	-21	
UnitEmptyErr	-22	
OpenErr	-23	
ClosErr	-24	
DRemovErr	-25	tried to remove an open driver
DInstErr	-26	DrvrInstall couldn't find driver in resources
AbortErr	-27	IO call aborted by KillIO
NotOpenErr	-28	Couldn't rd/wr/ctl/sts cause driver not opened

File System Errors

DirFulErr	-33	Directory full
DskFulErr	-34	disk full
NSVErr	-35	no such volume
IOErr	-36	I/O error (bummers)
BdNamErr	-37	there may be no bad names in the final system!
FNOpnErr	-38	File not open
EOFErr	-39	End of file
PosErr	-40	tried to position to before start of file (r/w)
MFulErr	-41	memory full(open) or file won't fit (load)
TMFOErr	-42	too many files open
FNFErr	-43	File not found
WPrErr	-44	diskette is write protected
FLckdErr	-45	file is locked
VLckdErr	-46	volume is locked
FBsyErr	-47	File is busy (delete)
DupFNErr	-48	duplicate filename (rename)
OpWrErr	-49	file already open with with write permission
ParamErr	-50	error in user parameter list
RFNumErr	-51	refnum error
GFPErr	-52	get file position error
VolOffLinErr	-53	volume not on line error (was Ejected)
PermErr	-54	permissions error (on file open)
VolOnLinErr	-55	drive volume already on-line at MountVol
NSDrvErr	-56	no such drive (tried to mount a bad drive num)
NoMacDskErr	-57	not a mac diskette (sig bytes are wrong)
ExtFSErr	-58	volume in question belongs to an external fs
FSDSErr	-59	file system 'system trouble' error: during rename the old entry was deleted but could not be restored . . .
BadMDBErr	-60	bad master directory block
WrPermErr	-61	write permissions error

Disk, Serial Ports, Clock Specific Errors

NoDriveErr	-64	drive not installed
OffLinErr	-65	r/w requested for an off-line drive

NoNybErr	-66	couldn't find 5 nybbles in 200 tries
NoAdrMkErr	-67	couldn't find valid addr mark
DataVerErr	-68	read verify compare failed
BadCkSmErr	-69	addr mark checksum didn't check
BadBtSlpErr	-70	bad addr mark bit slip nibbles
NoDtaMkErr	-71	couldn't find a data mark header
BadDCkSum	-72	bad data mark checksum
BadDBtSlp	-73	bad data mark bit slip nibbles
WrUnderRun	-74	write underrun occurred
CantStepErr	-75	step handshake failed
TkOBadErr	-76	track 0 detect doesn't change
InitIWMErr	-77	unable to initialize IWM
TwoSideErr	-78	tried to read 2nd side on a 1-sided drive
SpdAdjErr	-79	unable to correctly adjust disk speed
SeekErr	-80	track number wrong on address mark
SectNFErr	-81	sector number never found on a track
ClkRdErr	-85	unable to read same clock value twice
ClkWrErr	-86	time written did not verify
PRWrErr	-87	parameter ram written didn't read-verify
PRInitErr	-88	InitUtil found the parameter ram uninitialized
RcvrErr	-89	SCC receiver error (framing, parity, OR)
BreakRecd	-90	Break received (SCC)

Memory Manager Errors

MemFullErr	-108	Not enough room in heap zone
NilHandleErr	-109	Handle was NIL in HandleZone or other
memWZErr	-111	WhichZone failed (applied to free block)
memPurErr	-112	trying to purge a locked or non-purgable block
memAdrErr	-110	address was odd, or out of range
memAZErr	-113	Address in zone check failed
memPCErr	-114	Pointer Check failed
memBCErr	-115	Block Check failed
memSCErr	-116	Size Check failed

Resource Manager Errors (other than File System Errors)

ResNotFound	-192	Resource not found
ResFNotFound	-193	Resource file not found
AddResFailed	-194	AddResource failed
AddRefFailed	-195	AddReference failed
RmvResFailed	-196	RmveResource failed
RmvRefFailed	-197	RmveReference failed

Scrap Manager Errors

noScrapErr	-100	No scrap exists error
noTypeErr	-102	No object of that type in scrap

Application Errors

Errors -1024 to -4095 are reserved for use by the current application

some miscellaneous result codes

EvtNotEnb	1	event not enabled at PostEvent
NoEvtAvail	-1	no event available (GetOSEvent, OSEventAvail)

MACINTOSH USER EDUCATION

The Operating System Utilities: A Programmer's Guide /OSUTIL/UTIL

See Also: The Memory Manager: A Programmer's Guide
Programming Macintosh Applications in Assembly Language
Macintosh Packages: A Programmer's Guide
The Structure of a Macintosh Application

Modification History: First Draft Brent Davis 6/5/84

*** PRELIMINARY DRAFT 6/5/84 NOT FOR DISTRIBUTION *** ABSTRACT

This manual describes the Operating System Utilities, a set of routines and data types in the Operating System that perform generally useful operations such as manipulating pointers and handles, comparing strings, and reading the date and time.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the Operating System Utilities, a set of routines and data types in the Operating System that perform generally useful operations such as manipulating pointers and handles, comparing strings, and reading the date and time. *** Eventually this manual will become part of the comprehensive Inside Macintosh manual. ***

You should already be familiar with Lisa Pascal. Depending on which Operating System Utilities you're interested in using, you may also need to be familiar with other parts of the Toolbox or Operating System; where that's necessary, you're referred to the appropriate manuals.

This manual is intended to serve the needs of both Pascal and assembly-language programmers. Information of interest to assembly-language programmers only is isolated and labeled so that Pascal programmers can conveniently skip it.

This manual begins with a section on Operating System Utility data types; this is followed by a section describing the structure and function of parameter RAM, and then a description of the structure of Operating System queues and the data types associated with them. After that comes a detailed description of all Operating System Utility procedures and functions, their parameters, calling protocol, effects, side effects, and so on. Finally, there's a summary of the Operating System Utilities, for quick reference, followed by a glossary of terms used in this manual.

OPERATING SYSTEM UTILITY DATA TYPES

This section describes two data types of interest to users of the Operating System.

Four-Character Sequences

There are several places in the Operating System where you must specify a four-character sequence for something, such as for file types and application signatures (as described in The Structure of a Macintosh Application). The Pascal data type for such sequences is:

```
TYPE OSType = PACKED ARRAY [1..4] OF CHAR;
```

Result Codes

Many Operating System Utility routines, like most other Operating System routines, return a result code in addition to their normal results. This is an integer indicating whether the routine completed its task successfully or was prevented by some error condition. The type definition for result codes is

```
TYPE OSErr = INTEGER;
```

In the normal case that no error is detected, the result code is

```
CONST noErr = Ø;  {no error}
```

A nonzero result code signals an error. For example:

```
CONST qErr     =   -1;  {element not in specified queue}
      clkRdErr =  -85;  {unable to read same clock value twice}
```

PARAMETER RAM

Various settings, such as those specified by the user by means of the Control Panel desk accessory, need to remain in memory even when the Macintosh is off so they will still be present at the next system startup. This information is kept in parameter RAM, a specially dedicated section of RAM that's stored in the clock chip together with the current settings for the date and time. This chip runs on batteries, preserving the information whether the system is on or not.

You may find it necessary to read the values in parameter RAM or even change them (especially if you create a desk accessory like the Control Panel). Parameter RAM, however, contains its information in a highly compact, hard-to-access form. To make reading or changing this information less difficult, then, the contents of parameter RAM are copied into a more accessible 2Ø-byte section of low memory at system startup. The routines for reading and changing parameter RAM, as described in the "Operating System Utility Routines" section of this manual, go through this low-memory copy instead of directly accessing parameter RAM itself.

(note)

Some of the more useful values contained in the low-memory copy of parameter RAM can be accessed more easily by calling routines designed to return them: for instance, the Event Manager function DoubleTime returns the double-click time stored in the low-memory copy of parameter RAM. Each such routine is discussed in its appropriate manual.

Assembly-language note: The low memory copy of parameter RAM begins at the address sysParam; the various portions of the copy can be accessed by means of a series of global variables, all of which are listed in the summary. The contents of several of the more useful of these variables are copied into other global variables at system startup for even easier access: for instance, the auto-key threshold and rate, which are contained in the low-memory variable spKbd, are copied into the variables keyThresh and keyRepThresh, respectively. Each such variable is discussed in its appropriate manual.

At system startup, the date and time is also copied from the clock chip into its own low-memory location; this value is accessible by means of the Operating System Utility ReadDateTime, as described in the "Routines" section of this manual.

Assembly-language note: This value is accessible from assembly language in the global variable called time.

System Parameter Records

The 2Ø-byte low-memory copy of parameter RAM is represented internally by a system parameter record, which is defined as follows:

```
TYPE SysParmType =
       RECORD
         valid:    LongInt;  {validity status}
         portA:    INTEGER;  {modem port ("port A") configuration}
         portB:    INTEGER;  {prntr port ("port B") configuration}
         alarm:    LongInt;  {alarm setting}
         font:     INTEGER;  {default application font}
         kbdPrint: INTEGER;  {auto-key thresh/rate; prntr's port}
         volClik:  INTEGER;  {vol level; dbl-click/caret blink}
         misc:     INTEGER   {mouse scaling; boot disk;
                              menu blink}
       END;

     SysPPtr = ^sysParmType;
```

The structure of the system parameter record is illustrated in Figure 1.

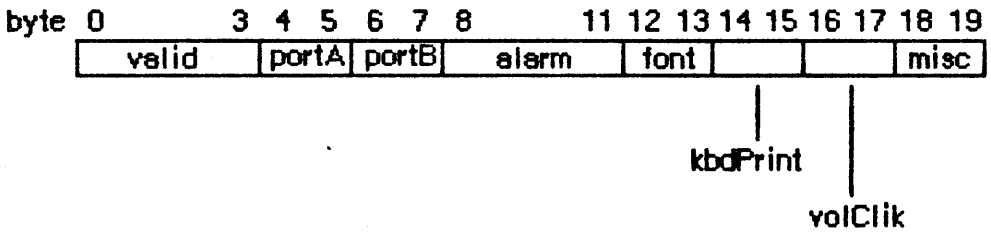


Figure 1. The System Parameter Record

The contents of the system parameter record's fields are as follows:

The low-order byte of the valid field contains the validity status of parameter RAM: whenever you write anything to parameter RAM, $A8 is stored in this byte if all the values being written are valid. If this byte does not contain $A8 at system startup, the contents of parameter RAM are initialized to certain standard default values.

```
Validity status             = $A8
Modem port configuration    = 96ØØ baud
                              8 data bits
                              2 stop bits
                              Ø parity
Printer port configuration = same as for modem port
Alarm setting               = Midnight, January 1, 19Ø4
Default application font    = 2 *** meaning? ***
Keyboard repeat threshold   = 24 ticks
Keyboard repeat rate        = 6 ticks
Printer port                = Ø *** meaning? ***
Volume control              = 3 *** meaning? ***
Double click time           = 32 ticks
Caret blink time            = 32 ticks
Misc2                       = $4C = 76 = Ø1ØØ11ØØ *** meaning? ***
```

The other three bytes of this field are unused. (See Figure 2.)

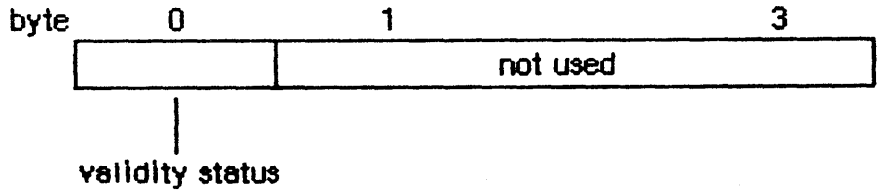


Figure 2. The Valid Field

The portA and portB fields contain the modem port and printer port configurations, respectively; these configurations contain information on the baud rates, data bits, stop bits and parity bits for the two ports. Port configurations are explained more fully in the Serial Driver manual *** which doesn't yet exist ***.

The alarm field contains the alarm setting in seconds since midnight, January 1, 19Ø4.

The font field contains the number of the default application font.

Bits Ø through 3 of the low-order byte of the kbdPrint field contain the auto-key threshold--that is, the length of time a key must be held down before it begins to repeat. This value is stored in four-tick units. Bits 4 through 7 of this byte contain the rate of the repeat itself; this value is stored in two-tick units. The high-order byte of this field contains a number designating whether the printer is connected to the modem port or the printer port. (See Figure 3.)

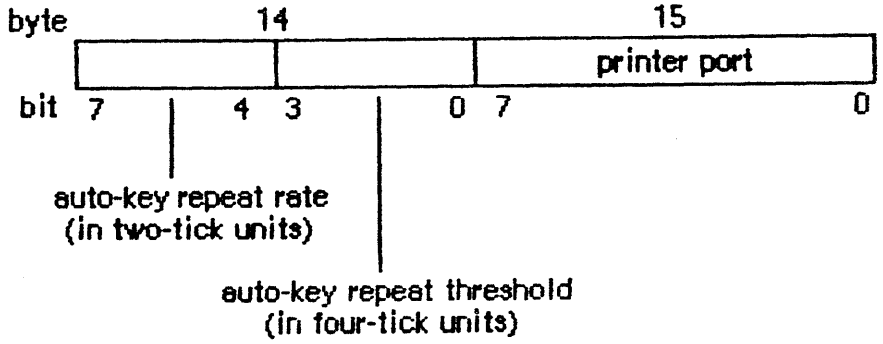


Figure 3. The KbdPrint Field

Bits Ø through 2 of the low-order byte of the volClik field contain the volume control *** in what units? ***; the remaining bits of this byte are unused. Bits Ø through 3 of the high-order byte of this field contain a number designating the greatest interval between a mouse-up and mouse-down that would qualify those two mouse clicks as a double click; this value is stored in four-tick units. Bits 4 through 7 of this byte contain a number designating the interval between blinks of the caret; this value is stored in four-tick units. (See Figure 4.)

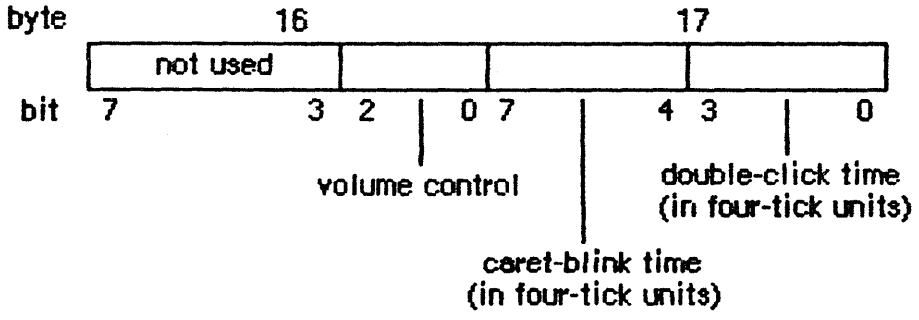


Figure 4. The volClik Field

The low-order byte of the misc field is unused. Bit 1 of the high-order byte designates the degree of mouse scaling *** define *** in effect. Bit 3 indicates whether the disk to be used to start up the system is in the internal or the external drive. Bits 4 and 5 contain a value that determines how many times a menu will blink when something is chosen from it. *** How? Check w/Menu Mgr manual. *** Bits Ø, 2, 6 and 7 of the high-order byte are not used. (See Figure 5.)

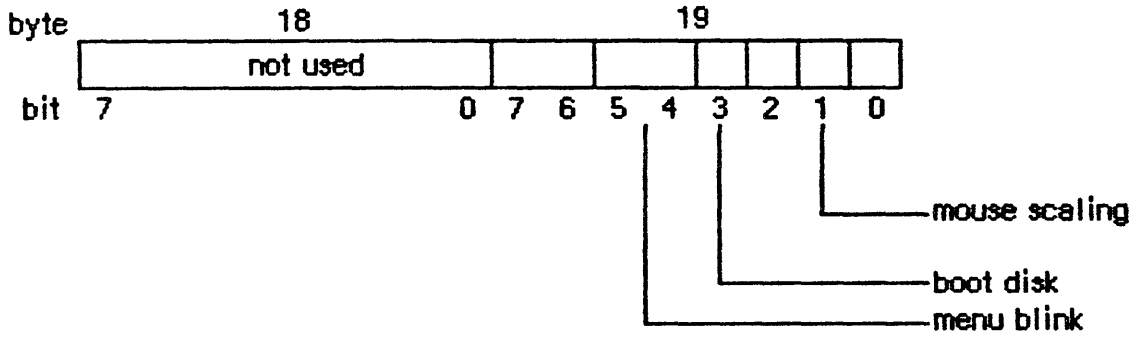


Figure 5. The Misc Field

Parameter Masks

The utilities that write to and read from parameter RAM expect a parameter mask as one of their parameters. This mask specifies which portions of parameter RAM are to be written or read. If, for instance, you wish to write a new default application font number to parameter RAM, you can supply the utility with a parameter mask that will restrict it to writing only to that one portion of parameter RAM.

The parameter mask is a long integer, the low-order 2Ø bits of which correspond to the 2Ø bytes of the system parameter record, as shown in Figure 6. *** Is Ø-31 numbering OK, or should it be 15 max? Also, fig is wider than 6". *** (The remaining bits of the parameter mask are unused.) A mask specifying any given field of the record is derived by setting the bits of the mask corresponding to the bytes that the field occupies in the record. For instance, since the font field of the system parameter record occupies bytes 12 and 13 of the record, it's specified by bits 12 and 13 of the mask. A 1 in each of those positions means that the parameter RAM utility routine applies to the portion of parameter RAM containing the number of the default application font.

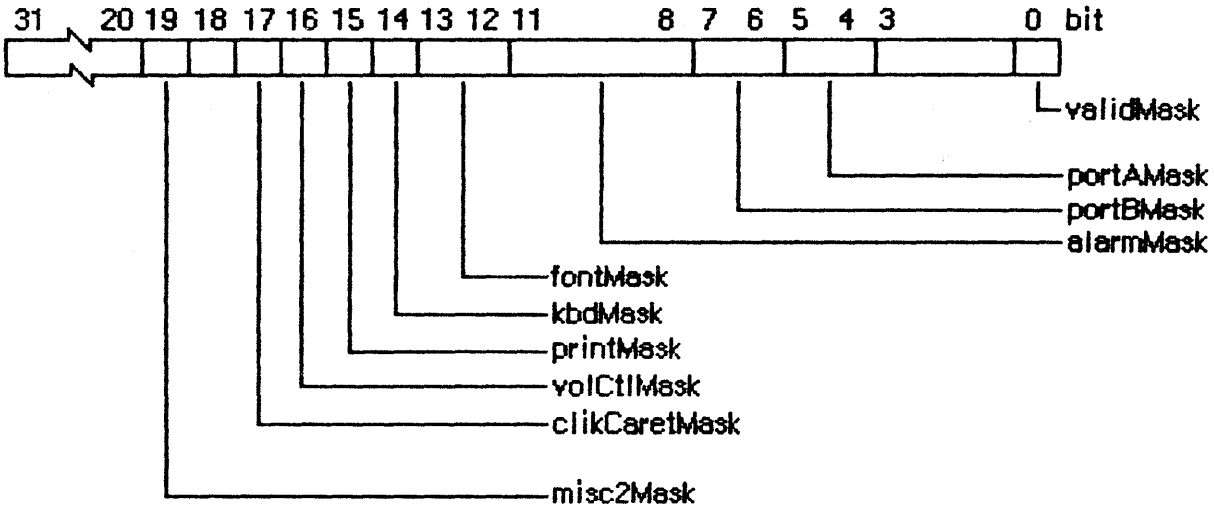


Figure 6. The Parameter Mask

Where there's more than one type of information in a field, only the bits corresponding to the desired information are set in the parameter mask. Information that occupies less than a byte cannot be specified by itself in the mask; the entire byte must be designated.

Each mask described so far is available as a predefined constant:

```
CONST validMask     = 1;       {validity status}
      portAMask     = 48;      {modem port configuration}
      portBMask     = 192;     {printer port configuration}
      alarmMask     = 384Ø;    {alarm setting}
      fontMask      = 12288;   {number of default application font}
      kbdMask       = 16384;   {auto-key threshold and rate}
      printMask     = 32768;   {printer's port}
      volCtlMask    = 65536;   {volume level}
      clikCaretMask = 131Ø72;  {dbl-click/caret-blink times}
      misc2Mask     = 524288;  {mouse scaling; boot disk; menu
                                blink} *** Change to miscMask. ***
```

There's also a predefined mask for designating all of parameter RAM:

```
CONST everyParam = 1Ø48575;  {all of parameter RAM}
```

You can form any mask you need by adding or subtracting these mask constants. For example, to specify both port A and port B configurations, use

```
portAMask + portBMask
```

For every portion of parameter RAM except the number of the default application font, use

everyParam - fontMask

OPERATING SYSTEM QUEUES

Some of the information used by the Operating System is stored in data structures called queues. A queue is a list of identically structured entries linked together by pointers. Queues are used to keep track of vertical retrace tasks, I/O requests, disk drives, events, and mounted volumes.

The structure of a standard Operating System queue is as follows:

```
TYPE QHdr = RECORD
              qFlags: INTEGER;  {queue flags}
              qHead:  QElemPtr; {first queue entry}
              qTail:  QElemPtr  {last queue entry}
            END;

     QHdrPtr = ^QHdr;
```

QFlags contains information that's different for each queue type. QHead points to the first entry in the queue, and qTail points to the last entry in the queue. The entries within each type of queue are different, since each type of queue contains different information. The Operating System uses the following variant record to access queue entries:

```
TYPE QTypes = (dummyType,
               vType,     {vertical retrace queue type}
               ioQType,   {I/O request queue type}
               drvQType,  {drive queue type}
               evType,    {event queue type}
               fsQType);  {volume-control-block queue type}

     QElem = RECORD
               CASE QTypes OF
                 (vblQElem: VBLTask);
                 (ioQElem:  ParamBlockRec);
                 (drvQElem: DrvQEl);
                 (evQElem:  EvQEl);
                 (vcbQElem: VCB)
             END;

     QElemPtr = ^QElem;
```

The exact structure of the entries in each type of Operating System queue is described in the manual that discusses that queue in detail. All entries in queues, though, regardless of the queue type, begin with a pointer to the next queue element and an integer designating the queue type.

Assembly-language note: The queue types are available to assembly-language programmers as the global constants vType, ioQType, evType, and fsQType. (There is no global constant corresponding to drvQType.) *** Check w/Roni; may have been fixed. ***

OPERATING SYSTEM UTILITY ROUTINES

This section describes all the Operating System Utility procedures and functions. They're presented in their Pascal form; for most routines, this is followed by a box containing information needed to use the routine from assembly language. Pascal programmers can just skip this box. For more information on using the Operating System utilities from assembly language, see Programming Macintosh Applications in Assembly Language.

Pointer and Handle Manipulation

FUNCTION HandToHand (VAR theHndl: Handle) : OsErr;

Trap macro	_HandToHand	
On entry	AØ:	theHndl (handle)
On exit	AØ:	theHndl (handle)
	DØ:	result code (integer)

HandToHand copies the information to which theHndl is a handle and returns a new handle to the copy in theHndl.

Result codes	noErr	No error
	memFullErr	Memory full *** Yes? ***

```
FUNCTION PtrToHand (srcPtr: Ptr; VAR dstHndl: Handle; size: LongInt) :
        OsErr;
```

Trap macro	_PtrToHand	
On entry	AØ:	srcPtr (pointer)
	DØ:	size (long integer)
On exit	A1:	dstHndl (handle)
	DØ:	result code (integer)

PtrToHand returns in dstHndl a newly created handle to the number of bytes specified by the size parameter, beginning at the location specified by srcPtr. *** Or does it return a new handle to a COPY of the info? ***

Result codes	noErr	No error
	memFullErr	Memory full *** Yes? ***

```
FUNCTION PtrToXHand (srcPtr: Ptr; dstHndl: Handle; size: LongInt) :
        OsErr;
```

Trap macro	_PtrToXHand	
On entry	AØ:	srcPtr (pointer)
	A1:	dstHndl (handle)
	DØ:	size (long integer)
On exit	DØ:	result code (integer)

PtrToXHand takes an existing handle specified by dstHndl and makes it a handle to the number of bytes specified by the size parameter, beginning at the location specified by srcPtr. *** Or does it return a new handle to a COPY of the info? ***

Result codes	noErr	No error
	*** Other error conditions? ***	

```
FUNCTION HandAndHand (aHndl,bHndl: Handle) : OsErr;
```

Trap macro	_HandAndHand	
On entry	AØ:	aHndl (handle)
	A1:	bHndl (handle)
On exit	DØ:	result code (integer)

HandAndHand concatenates the information to which aHndl is a handle onto the end of the information to which bHndl is a handle.

Result codes	noErr	No error
	*** Other error conditions? ***	

```
FUNCTION PtrAndHand (pntr: Ptr; hndl: Handle; size: LongInt) : OsErr;
```

Trap macro	_PtrAndHand	
On entry	AØ:	pntr (pointer)
	A1:	hndl (handle)
	DØ:	size (long integer)
On exit	DØ:	result code (integer)

PtrAndHand takes the number of bytes specified by the size parameter, beginning at the location specified by pntr, and concatenates that information onto the end of the information to which hndl is a handle.

Result codes	noErr	No error
	*** Other error conditions? ***	

String Comparison

Assembly-language note: The trap macros for these utility

routines have optional arguments corresponding to the Pascal flags associated with those routines. When present, such an argument sets a certain bit of the routine trap word; this is equivalent to setting the corresponding Pascal flag to TRUE. The trap macros for these routines appear with all the possible permutations of arguments. Whichever permutation you use, you must type it exactly as shown.

```
FUNCTION EqualString (aStr,bStr: Str255; case,marks: BOOLEAN) :
        BOOLEAN;
```

Trap macro	_CmpString	
	_CmpString ,MARKS	(sets bit 9)
	_CmpString ,CASE	(sets bit 1Ø)
	_CmpString ,MARKS,CASE	(sets bits 9 and 1Ø)
On entry	AØ: aStr (pointer to string)	
	A1: bStr (pointer to string)	
	DØ: high-order word:	length of string pointed to by aStr
	low-order word:	length of string pointed to by bStr
On exit	DØ: Ø if strings equal	
	1 if strings not equal	

EqualString compares the two given strings for equality *** on what basis? ASCII values? ***. If the marks parameter is TRUE, diacritical marks are ignored during the comparison; if the case parameter is TRUE, uppercase characters are distinguished from the corresponding lowercase characters. The function returns TRUE if the strings are equal.

(note)

See also the International Utilities Package function IUEqualString, as described in the Macintosh Packages manual.

```
PROCEDURE UprString (VAR theString: Str255; marks: BOOLEAN);
```

Trap macro	_UprString	
	_UprString ,MARKS	(sets bit 9)
On entry	A0:	theString (pointer to string)
	D0:	length of string pointed to by theString

UprString converts any lowercase letters in the given string to uppercase characters, returning the converted string in theString. Diacritical marks are ignored during the comparison if the marks parameter is TRUE.

Date and Time Operations

The following utilities are for reading and setting the date and time stored in the clock chip. Reading the date and time are fairly common operations; setting them is somewhat rarer, but could be necessary for implementing a desk accessory like the standard Control Panel.

Date and time are represented internally by a date/time record. Date/time records are defined as follows:

```
TYPE DateTimeRec =
       RECORD
         year:      INTEGER;  {four-digit year}
         month:     INTEGER;  {1 to 12 for January to December}
         day:       INTEGER;  {1 to 31}
         hour:      INTEGER;  {0 to 23}
         minute:    INTEGER;  {0 to 59}
         second:    INTEGER;  {0 to 59}
         dayOfWeek: INTEGER;  {1 to 7 for Sunday to Saturday}
       END;
```

```
FUNCTION ReadDateTime (VAR secs: LongInt) : OsErr;
```

Trap macro	_ReadDateTime	
On exit	A0:	secs (long integer)
	D0:	result code (integer)

ReadDateTime returns in secs the number of seconds between midnight, January 1, 1904 and the time that the function was called.

Assembly-language note: This value is accessible from assembly language in the global variable called time.

If you wish, you can convert the value returned by ReadDateTime to a date/time record by calling the Operating System Utility procedure Secs2Date.

(note)
Passing the value returned by ReadDateTime to the International Utilities Package procedure IUDateString or IUTimeString will yield a string representing the corresponding date or time of day, respectively.

Result codes	noErr	No error
	clkRdErr	Unable to read same clock value twice *** Explain. ***

```
FUNCTION SetDateTime (secs: LongInt) : OsErr;
```

Trap macro	_SetDateTime	
On entry	DØ:	secs (long integer)
On exit	DØ:	result code (integer)

SetDateTime takes a number of seconds since midnight, January 1, 19Ø4 as specified by secs and writes it to the clock chip as the current date and time.

(warning)
Any attempt to write a time earlier than midnight, January 1, 19Ø4 will result in an error.

Assembly-language note: This procedure also updates the global variable called time to the value of the secs parameter.

Result codes	noErr	No error
	clkWrErr	Tried to write an invalid time

PROCEDURE Date2Secs (date: DateTimeRec; VAR secs: LongInt);

Trap macro	_Date2Secs	
On entry	A0:	pointer to date/time record
On exit	D0:	secs (long integer)

Date2Secs takes the given date/time record, converts it to the corresponding number of seconds elapsed since midnight, January 1, 1904, and returns the result in the secs parameter.

PROCEDURE Secs2Date (secs: LongInt; VAR date: DateTimeRec);

Trap macro	_Secs2Date	
On entry	D0:	secs (long integer)
On exit	A0:	pointer to date/time record

Secs2Date takes a number of seconds elapsed since midnight, January 1, 1904 as specified by the secs parameter, converts it to the corresponding date and time, and returns the corresponding date/time record in the date parameter.

PROCEDURE GetTime (VAR date: DateTimeRec); [Pascal only]

GetTime takes the number of seconds elapsed since midnight, January 1, 1904 (obtained by calling ReadDateTime), converts that value into a date and time (by calling Secs2Date), and returns the result in the date parameter.

Assembly-language note: From assembly language, you can just call ReadDateTime and Secs2Date directly.

PROCEDURE SetTime (date: DateTimeRec); [Pascal only]

SetTime takes the date and time specified by the date parameter, converts it into the corresponding number of seconds elapsed since midnight, January 1, 1904 (by calling Date2Secs), and then writes that value to the clock chip as the current date and time (by calling SetDateTime).

Assembly-language note: From assembly language, you can just call Date2Secs and SetDateTime directly.

Parameter RAM Operations

The following three utilities are used for reading from and writing to parameter RAM. Figure 7 illustrates the function of these three utilities; further details are given below and earlier in the "Parameter RAM" section.

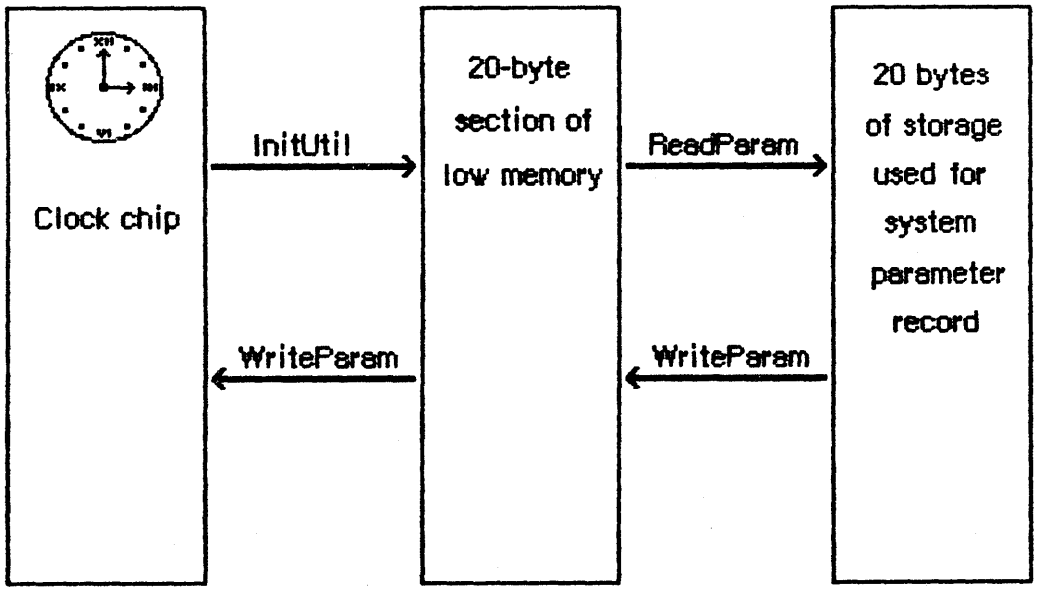


Figure 7. Function of Parameter RAM Utilities

FUNCTION InitUtil : OsErr;

Trap macro _InitUtil

On exit	DØ:	result code (integer)

InitUtil copies the contents of parameter RAM into 2Ø-bytes of low memory; it also copies the date and time from the clock chip into the low-memory location accessed by ReadDateTime.

Assembly-language note: InitUtil copies the date and time into the global variable called time.

This utilitiy is called for you at system startup; you will probably never need to call it yourself.

If parameter RAM contains any invalid values when this utility is called, then an error is returned as the result code, and the default values given earlier in the "Parameter RAM" section are read into the low-memory locations for parameter RAM and the current date and time.

Result codes	noErr	No invalid values in parameter RAM
	prInitErr	Invalid values in parameter RAM

```
FUNCTION ReadParam (prmsToRead: SysPPtr; paramMask: LongInt) : OsErr;
```

Trap macro	_ReadParam	
On entry	AØ:	prmsToRead (pointer to system parameter record)
	DØ:	paramMask (long integer)
On exit	DØ:	result code (integer)

ReadParam is primarily used for examining portions of parameter RAM before changing them. It copies portions of parameter RAM into 2Ø bytes of storage to be used for a system parameter record; the storage is pointed to by prmsToRead.

Assembly-language note: There is no need for assembly-language

programmers to use 2Ø bytes of storage in order to read from parameter RAM; you can directly access the global variables representing the various portions of the 2Ø-byte section of low memory. These variables begin at the address sysParam and are given in the summary.

ReadParam takes a parameter mask (paramMask) as one of its parameters; this mask should designate only those portions of parameter RAM that you wish to read, as described earlier in the "Parameter RAM" section.

Result codes	noErr	No error
	*** What are the error conditions? ***	

FUNCTION WriteParam (prmsToWrite: SysPPtr; paramMask: LongInt) : OsErr;

Trap macro	_WriteParam	
On entry	AØ:	prmsToWrite (pointer to system parameter record)
	DØ:	paramMask (long integer)
On exit	DØ:	result code (integer)

WriteParam updates the low-memory copy of parameter RAM and then goes on to update parameter RAM itself by writing the changes to the clock chip. The prmsToWrite parameter is a pointer to 2Ø bytes of storage containing a system parameter record that you create; this record describes what parameter RAM and its low-memory copy will look like after they're updated. If the system parameter record contains invalid values, then an error will be returned as the result code.

Assembly-language note: There is no need for assembly-language programmers to use 2Ø bytes of storage in order to write to parameter RAM; simply designate as the storage area the 2Ø-byte section of low memory itself, which begins at the address sysParam.

WriteParam takes a parameter mask (paramMask) as one of its parameters; this parameter mask should designate only those portions of parameter RAM that will actually be changed, as described earlier in the "Parameter RAM" section.

Result codes	noErr	No error
	prWrErr	Tried to write from an invalid

system parameter record

Queue Manipulation

This section describes two utilities for adding elements to or deleting elements from a queue. Most programmers won't need to use these utilities, since Operating System or Toolbox units that deal with queues take care of queue manipulation for you.

```
PROCEDURE Enqueue (qElement: QElemPtr; theQ: QHdrPtr);
```

Trap macro	_Enqueue	
On entry	A0:	qElement (pointer)
	A1:	theQ (pointer)

Enqueue adds the queue element pointed to by qElement to the queue pointed to by theQ. *** Mightn't there be errors, like maybe memFullErr or a queue type error? ***

```
FUNCTION Dequeue (qElement: QElemPtr; theQ: QHdrPtr) : OsErr;
```

Trap macro	_Dequeue	
On entry	A0:	qElement (pointer)
	A1:	theQ (pointer)
On exit	D0:	result code (integer)

Dequeue deletes the queue element pointed to by qElement from the queue pointed to by theQ.

Result codes	noErr	No error
	qErr	Element not in specified queue

Dispatch Table Utilities

This section describes a pair of utility routines for manipulating the dispatch table, which is described more fully in the Memory Manager manual. *** Currently it's described in the manual Programming Macintosh Applications in Assembly Language. ***

```
FUNCTION GetTrapAddress (trapNum: INTEGER) : LongInt;
```

Trap macro	_GetTrapAddress
On entry	DØ: trapNum (integer)
On exit	AØ: address of routine

GetTrapAddress returns the address of a routine currently installed in the dispatch table under the trap number designated by trapNum. A list of these numbers is given in Appendix A *** (doesn't yet exist) ***.

Assembly-language note: On entry, only the low-order nine bits of DØ are used; the rest of the register is ignored. This allows you to use a full trap word, created with the corresponding trap macro, to specify the trap number. On exit, the contents of register DØ are not preserved.

One use for GetTrapAddress is to save time in critical sections of your program by calling an OS or Toolbox routine directly, avoiding the overhead of a normal trap dispatch.

Assembly-language note: When you use this technique to bypass the Trap Dispatcher, you don't get the extra level of register saving. The routine itself will follow Lisa Pascal conventions and preserve A2-A6 and D3-D7, but if you want any other registers preserved across the call you have to save and restore them yourself.

You can also use GetTrapAddress when you want to intercept calls to an Operating System or Toolbox routine and do some pre- or postprocessing of your own. Before installing your own version of the routine in the dispatch table, you can call GetTrapAddress to get the address of the

original and save it somewhere for later use. The new version of the routine can then use this saved address to call the original version.

(warning)
A number of ROM routines have already been patched with corrected versions in RAM; for the system to work properly, certain of these patched routines shouldn't be replaced with versions of your own. It's recommended that you don't replace any of the existing routines unless you're sure you know what you're doing.

PROCEDURE SetTrapAddress (trapAddr: LongInt; trapNum: INTEGER);

Trap macro _SetTrapAddress

On entry A0: trapAddr (address)
D0: trapNum (integer)

SetTrapAddress installs in the dispatch table a routine whose address is trapAddr; this routine is installed under the trap number designated by trapNum.

Assembly-language note: On entry, only the low-order nine bits of D0 are used; the rest of the register is ignored. This allows you to use a full trap word, created with the corresponding trap macro, to specify the trap number. On exit, the contents of register D0 are not preserved.

(warning)
A number of ROM routines have already been patched with corrected versions in RAM; for the system to work properly, certain of these patched routines shouldn't be replaced with versions of your own. It's recommended that you don't replace any of the existing routines unless you're sure you know what you're doing.

You can also use SetTrapAddress to install your own routines in unused slots in the dispatch table, allowing them to be called via the trap mechanism like Operating System and Toolbox routines.

Miscellaneous Utilities

PROCEDURE Delay (numTicks: LongInt; VAR finalTicks: LongInt);

Trap macro	_Delay	
On entry	AØ:	numTicks (long integer)
On exit	DØ:	finalTicks (long integer)

Delay simply causes the system to wait for the number of ticks specified by numTicks; it then returns in the finalTicks parameter the total number of ticks from the last system startup to the end of the delay.

Assembly-language note: The current number of elapsed ticks since system startup is contained in the global variable called ticks; on exit from this procedure, register DØ contains the value of this global variable as measured at the end of the delay. Calling this procedure sends an interrupt priority level of Ø to the processor--that is, no interrupting devices will be ignored. See the Device Manager manual *** (doesn't yet exist) *** for further details on interrupts.

PROCEDURE SysBeep (duration: INTEGER);

SysBeep causes the system to beep for the number of seconds specified by the duration parameter.

(note)
Unlike all other Operating System Utilities, this procedure is stack-based.

SUMMARY OF THE OPERATING SYSTEM UTILITIES

Constants

```
CONST { Parameter masks }

      validMask     = 1;       {validity status}
      portAMask     = 48;      {modem port configuration}
      portBMask     = 192;     {printer port configuration}
      alarmMask     = 384Ø;    {alarm setting}
      fontMask      = 12288;   {number of default application font}
      kbdMask       = 16384;   {auto-key threshold and rate}
      printMask     = 32768;   {printer's port}
      volCtlMask    = 65536;   {volume level}
      clikCaretMask = 131Ø72;  {dbl-click/caret-blink times}
      misc2Mask     = 524288;  {mouse scaling; boot disk; menu
                                blink}

      { Result codes }

      noErr     =  Ø;   {no error}
      qErr      = -1;   {element not in specified queue}
      clkRdErr  = -85;  {unable to read same clock value twice}
      clkWrErr  = -86;  {tried to write an invalid time}
      prWrErr   = -87;  {tried to write from an invalid sysParmType}
      prInitErr = -88;  {validity status is not $A8}
```

Data Types

```
TYPE OSType = PACKED ARRAY [1..4] OF CHAR;

     OSErr = INTEGER;

     SysParmType =
       RECORD
         valid:    LongInt;  {validity status}
         portA:    INTEGER;  {modem port ("port A") configuration}
         portB:    INTEGER;  {prntr port ("port B") configuration}
         alarm:    LongInt;  {alarm setting}
         font:     INTEGER;  {default application font}
         kbdPrint: INTEGER;  {auto-key thresh/rate; prntr's port}
         volClik:  INTEGER;  {vol level; dbl-click/caret blink}
         misc:     INTEGER   {mouse scaling; boot disk;
                              menu blink}
       END;

     SysPPtr = ^sysParmType;

     QHdr = RECORD
              qFlags: INTEGER;  {queue flags}
```

```
        qHead:  QElemPtr; {first queue entry}
        qTail:  QElemPtr  {last queue entry}
      END;

QHdrPtr = ^QHdr;

QTypes = (dummyType,
          vType,     {vertical retrace queue type}
          ioQType,   {I/O request queue type}
          drvQType,  {drive queue type}
          evType,    {event queue type}
          fsQType);  {volume-control-block queue type}

QElem = RECORD
          CASE QTypes OF
            (vblQElem: VBLTask);
            (ioQElem:  ParamBlockRec);
            (drvQElem: DrvQEl);
            (evQElem:  EvQEl);
            (vcbQElem: VCB)
        END;

QElemPtr = ^QElem;

DateTimeRec =
  RECORD
    year:      INTEGER;  {four-digit year}
    month:     INTEGER;  {1 to 12 for January to December}
    day:       INTEGER;  {1 to 31}
    hour:      INTEGER;  {Ø to 23}
    minute:    INTEGER;  {Ø to 59}
    second:    INTEGER;  {Ø to 59}
    dayOfWeek: INTEGER;  {1 to 7 for Sunday to Saturday}
  END;
```

Routines

Pointer and Handle Manipulation

```
FUNCTION  HandToHand  (VAR theHndl: Handle) : OsErr;
FUNCTION  PtrToHand   (srcPtr: Ptr; VAR dstHndl: Handle; size:
                       LongInt) : OsErr;
FUNCTION  PtrToXHand  (srcPtr: Ptr; dstHndl: Handle; size: LongInt) :
                       OsErr;
FUNCTION  HandAndHand (aHndl, bHndl: Handle) : OsErr;
FUNCTION  PtrAndHand  (ptr: Ptr; hndl: Handle; size: LongInt) : OsErr;
```

String Comparison

```
FUNCTION  EqualString (aStr,bStr: Str255; case,marks: BOOLEAN) :
                       BOOLEAN;
PROCEDURE UprString   (VAR theString: Str255; marks: BOOLEAN);
```

Date and Time Operations

```
FUNCTION  ReadDateTime (VAR secs: LongInt) : OsErr;
FUNCTION  SetDateTime  (secs: LongInt) : OsErr;
PROCEDURE Date2Secs    (date: DateTimeRec; VAR secs: LongInt);
PROCEDURE Secs2Date    (secs: LongInt; VAR date: DateTimeRec);
PROCEDURE GetTime      (VAR date: DateTimeRec);  [Pascal only]
PROCEDURE SetTime      (date: DateTimeRec);  [Pascal only]
```

Parameter RAM Operations

```
FUNCTION  InitUtil : OsErr;
FUNCTION  ReadParam  (prmsToRead: SysPPtr; paramMask: LongInt) :
                      OsErr;
FUNCTION  WriteParam (prmsToWrite: SysPPtr; paramMask: LongInt) :
                      OsErr;
```

Queue Manipulation

```
PROCEDURE Enqueue (qElement: QElemPtr; theQ: QHdrPtr);
FUNCTION  Dequeue (qElement: QElemPtr; theQ: QHdrPtr) : OsErr;
```

Dispatch Table Utilities

```
PROCEDURE SetTrapAddress (trapAddr: LongInt; trapNum: INTEGER);
FUNCTION  GetTrapAddress (trapNum: INTEGER) : LongInt;
```

Miscellaneous Utilities

```
PROCEDURE Delay (numTicks: LongInt; VAR finalTicks: LongInt);
PROCEDURE SysBeep (duration: INTEGER);
```

Assembly-Language Information

Constants

```
; Result codes

noErr       .EQU   Ø    ;no error
qErr        .EQU  -1    ;element not in specified queue
clkRdErr    .EQU  -85   ;unable to read same clock value twice
clkWrErr    .EQU  -86   ;tried to write an invalid time
prWrErr     .EQU  -87   ;tried to write from an invalid sysParmType
prInitErr   .EQU  -88   ;validity status not $A8

; Queue types

vType               VBL queue element
ioQType             I/O queue element
evType              Event queue element
fsQType             File system VCB element
```

Queue Data Structure

```
qFlags              Queue flags
qHead               Pointer to first queue entry
qTail               Pointer to last queue entry
```

Queue Element Data Structure

```
qLink               Pointer to next queue element
qType               Queue type
```

Date/Time Record Data Structure

```
dtYear              Four-digit year
dtMonth             1 to 12 for January to December
dtDay               1 to 31
dtHour              Ø to 23
dtMinute            Ø to 59
dtSecond            Ø to 59
dtDayOfWeek         1 to 7 for Sunday to Saturday
```

Variables

Name	Size	Contents
spValid	1 byte	Validation field ($A8)
spPortA	2 bytes	Port A configuration
spPortB	2 bytes	Port B configuration

spAlarm	4 bytes	Alarm time
spFont	2 bytes	Default font ID
spKbd	1 bytes	Keyboard repeat threshold and rate
spPrint	1 byte	Print stuff
spVolCtl	1 byte	Volume control
spClikCaret	1 byte	Double click and caret blink times
spMisc2	1 byte	Mouse scaling, boot disk, menu blink
time	4 bytes	Seconds since midnight, January 1, 19Ø4
ticks	4 bytes	Ticks since last system startup

GLOSSARY

clock chip:

date/time record:

dispatch table:

parameter mask:

parameter RAM:

queue:

result code:

system parameter record:

THIS SECTION
INTENTIONALLY
LEFT BLANK.

**WHEN AVAILABLE,
IT WILL BE SUPPLIED
AS PART OF THE
MACINTOSH SUPPLEMENT.**

MACINTOSH USER EDUCATION

The Structure of a Macintosh Application /STRUCTURE/STRUCT

See Also: Macintosh User Interface Guidelines
Inside Macintosh: A Road Map
The Segment Loader: A Programmer's Guide
Putting Together a Macintosh Application

Modification History: First Draft (ROM 7) Caroline Rose 2/8/84

ABSTRACT

This manual describes the overall structure of a Macintosh application program, including its interface with the Finder.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes the overall structure of a Macintosh application program, including its interface with the Finder. *** Right now it describes only the Finder interface; the rest will be filled in later. Eventually it will become part of a comprehensive manual describing the entire Toolbox and Operating System. ***

(hand)
This information in this manual applies to version 7 of the Macintosh ROM and version 1.Ø of the Finder.

You should already be familiar with the following:

- The details of the User Interface Toolbox, the Macintosh Operating System, and the other routines that your application program may call. For a list of all the technical documentation that provides these details, see Inside Macintosh: A Road Map.

- The Finder, which is described in the Macintosh owner's guide.

This manual doesn't cover the steps necessary to create an application's resources or to compile, link, and execute the application program. These are discussed in the manual Putting Together a Macintosh Application.

The manual begins with sections that describe the Finder interface: signatures and file types, used for identification purposes; application resources that provide icon and file information to the Finder; and the mechanism that allows documents to be opened or printed from the Finder.

*** more to come ***

Finally, there's a glossary of terms used in this manual.

SIGNATURES AND FILE TYPES

Every application must have a unique signature by which the Finder can identify it. The signature can be any four-character sequence not being used for another application on any currently mounted volume (except that it can't be one of the standard resource types). To ensure uniqueness on all volumes, your application's signature must be assigned by Macintosh Technical Support.

Signatures work together with file types to enable the user to open or print a document (any file created by an application) from the Finder. When the application creates a file, it sets the file's creator and file type. Normally it sets the creator to its signature and the file type to a four-character sequence that identifies files of that type. When the user asks the Finder to open or print the file, the Finder

starts up the application whose signature is the file's creator and passes the file type to the application along with other identifying information, such as the file name. (More information about this process is given below under "Opening and Printing Documents from the Finder".)

An application may create its own special type or types of files. Like signatures, file types must be assigned by Macintosh Technical Support to ensure uniqueness. When the user chooses Open from an application's File menu, the application will display (via the Standard File Package) the names of all files of a given type or types, regardless of which application created the files. Having a unique file type for your application's special files ensures that only the names of those files will be displayed for opening.

(hand)

> Signatures and file types may be strange, unreadable combinations of characters; they're never seen by end users of Macintosh.

Applications may also create existing types of files. There might, for example, be one that merges two MacWrite documents into a single document. In such cases, the application should use the same file type as the original application uses for those files. It should also specify the original application's signature as the file's creator; that way, when the user asks the Finder to open or print the file, the Finder will call on the original application to perform the operation. To learn the signatures and file types used by existing applications, check with Macintosh Technical Support.

Files that consist only of text--a stream of characters, with Return characters at the ends of paragraphs or short lines--should be given the file type 'TEXT'. This is the type that MacWrite gives to text-only files it creates, for example. If your application uses this file type, its files will be accepted by MacWrite and it in turn will accept MacWrite text-only files (likewise for any other application that deals with 'TEXT' files). Your application can give its own signature as the file's creator if it wants to be called to open or print the file when the user requests this from the Finder.

For files that aren't to be opened or printed from the Finder, as may be the case for certain data files created by the application, the signature should be set to '????' (and the file type to whatever is appropriate).

FINDER-RELATED RESOURCES

To establish the proper interface with the Finder, every application's resource file must specify the signature of the application along with data that provides version information. In addition, there may be resources that provide information about icons and files related to the application. All of these Finder-related resources are described

below, followed by a comprehensive example and (for interested programmers) the exact formats of the resources.

Version Data

Your application's resource file must contain a special resource that has the signature of the application as its resource type. This resource is called the version data of the application. The version data is typically a string that gives the name, version number, and date of the application, but it can in fact be any data at all. The resource ID of the version data is Ø by convention.

As described in detail in Putting Together a Macintosh Application, part of the process of installing an application on the Macintosh is to set the creator of the file that contains the application. You set the creator to the application's signature, and the Finder copies the corresponding version data into a resource file named Desktop. (The Finder doesn't display this file on the Macintosh desktop, to ensure that the user won't tamper with it.)

(hand) -
Additional, related resources may be copied into the Desktop file; see "Bundles" below for more information. The Desktop file also contains folder resources, one for each folder on the volume.

Icons and File References

For each application, the Finder needs to know:

- The icon to be displayed for the application on the desktop, if different from the Finder's default icon for applications (see Figure 1).
- If the application creates any files, the icon to be displayed for each type of file it creates, if different from the Finder's default icon for documents.
- What files, if any, must accompany the application when it's transferred to another volume.

Figure 1. The Finder's Default Icons

The Finder learns this information from resources called file references in the application's resource file. Each file reference contains a file type and an ID number, called a local ID, that

identifies the icon to be displayed for that type of file. (The local ID is mapped to an actual resource ID as described under "Bundles" below.) Any file reference may also include the name of a file that must accompany the application when it's transferred to another volume.

The file type for the application itself is 'APPL'. This is the file type in the file reference that designates the application's icon. You also specify it as the application's file type at the same time that you specify its creator--the first time you install the application on the Macintosh.

The ID number in a file reference corresponds not to a single icon but to an icon list in the application's resource file. The icon list consists of two icons: the actual icon to be displayed on the desktop, and a mask consisting of that icon's outline filled with black (see Figure 2). *** For existing types of files, there's currently no way to direct the Finder to use the original application's icon for that file type. ***

Figure 2. Icon and Mask

Bundles

A bundle in the application's resource file groups together all the Finder-related resources. It specifies the following:

- The application's signature and the resource ID of its version data
- A mapping between the local IDs for icon lists (as specified in file references) and the actual resource IDs of the icon lists in the resource file
- Local IDs for the file references themselves and a mapping to their actual resource IDs

The first time you install the application on the Macintosh, you set its "bundle bit", and the Finder copies the version data, bundle, icon lists, and file references from the application's resource file into the Desktop file. *** (The setting of the bundle bit will be covered in the next version of Putting Together a Macintosh Application.) *** If there are any resource ID conflicts between the icon lists and file references in the application's resource file and those in Desktop, the Finder will change those resource IDs in Desktop. The Finder does this same resource copying and ID conflict resolution when you transfer an application to another volume.

(hand)
The local IDs are needed only for use by the Finder.

An Example

Suppose you've written an application named SampWriter. The user can create a unique type of document from it, and you want a distinctive icon for both the application and its documents. The application's signature, as assigned by Macintosh Technical Support, is 'SAMP'; the file type assigned for its documents is 'SAMF'. Furthermore, a file named 'TgFil' should accompany the application when it's transferred to another volume. You would include the following resources in the application's resource file:

Resource	Resource ID	Contents
Version data with resource type 'SAMP'	Ø	The string 'SampWriter Version 1 -- 2/1/84'
Icon list	128	The icon for the application The icon's mask
Icon list	129	The icon for documents The icon's mask
File reference	128	File type 'APPL' Local ID Ø for the icon list
File reference	129	File type 'SAMF' Local ID 1 for the icon list File name 'TgFil'
Bundle	128	Signature 'SAMP' Resource ID Ø for the version data For icon lists, the mapping: local ID Ø --> resource ID 128 local ID 1 --> resource ID 129 For file references, the mapping: local ID Ø --> resource ID 128 local ID 1 --> resource ID 129

(hand)
See the manual Putting Together a Macintosh Application for information about how to include these resources in a resource file.

The file references in this example happen to have the same local IDs and resource IDs as the icon lists, but any of these numbers can be different. Different resource IDs can be given to the file references, and the local IDs specified in the mapping for file references can be whatever desired.

Formats of Finder-Related Resources

The resource type for an application's version data is the signature of the application, and the resource ID is Ø by convention. The resource data can be anything at all; typically it's a string giving the name, version number, and date of the application.

The resource type for an icon list is 'ICN#'. The resource data simply consists of the icons, 128 bytes each.

The resource type for a file reference is 'FREF'. The resource data has the format shown below.

Number of bytes	Contents
4 bytes	File type
2 bytes	Local ID for icon list
1 byte	Length of following file name in bytes; Ø if none
n bytes	Optional file name

The resource type for a bundle is 'BNDL'. The resource data has the format shown below. The format is more general than needed for Finder-related purposes because bundles will be used in other ways in the future.

Number of bytes	Contents
4 bytes	Signature of the application
2 bytes	Resource ID of version data
2 bytes	Number of resource types in bundle minus 1
For each resource type:	
4 bytes	Resource type
2 bytes	Number of resources of this type minus 1
For each resource:	
2 bytes	Local ID
2 bytes	Actual resource ID

A bundle used for establishing the Finder interface contains the two resource types 'ICN#' and 'FREF'.

OPENING AND PRINTING DOCUMENTS FROM THE FINDER

When the user selects a document and tries to open or print it from the Finder, the Finder starts up the application whose signature is the document file's creator. An application may be selected along with one or more documents for opening (but not printing); in this case, the Finder starts up that application. If the user selects more than one document for opening without selecting an application, the files must have the same creator. If more than one document is selected for printing, the Finder starts up the application whose signature is the first file's creator (that is, the first one selected if they were selected by Shift-clicking, or the top left one if they were selected

by dragging a rectangle around them).

Any time the Finder starts up an application, it passes along information via the "Finder information handle" in the application parameter area (as described in the Segment Loader manual). Pascal programmers can call the Segment Loader procedure GetAppParms to get the Finder information handle. For example, if applParam is declared as type Handle, the call

```
GetAppParms(applName, applRefNum, applParam)
```

returns the Finder information handle in applParam. The Finder information has the following format:

Number of bytes	Contents
2 bytes	Ø if open, 1 if print
2 bytes	Number of files to open or print (Ø if none)
For each file:	
2 bytes	Volume reference number of volume containing the file
4 bytes	File type
1 byte	File's version number (typically Ø)
1 byte	Ignored
1 byte	Length of following file name in bytes
n bytes	Characters of file name (if n is even, add an extra byte)

The files are listed in order of the appearance of their icons on the desktop, from left to right and top to bottom. The file names don't include a volume prefix. An extra byte is added to any name of even length so that the entry for the next name will begin on a word boundary.

Every application that opens or prints documents should look at this information to determine what to do when the Finder starts it up. If the number of files is Ø, the application should start up with an untitled document on the desktop. If a file or files are specified for opening, it should start up with those documents on the desktop. If only one document can be open at a time but more than one file is specified, the application should open the first one and ignore the rest. If the application doesn't recognize a file's type (which can happen if the user selected the application along with another application's document), it may want to open the file anyway and check its internal structure to see if it's a compatible type. The response to an unacceptable type of file should be an alert box that shows the file name and says that the document can't be opened.

If a file or files are specified for printing, the application should print them in turn, preferably without doing its entire start-up sequence. For example, it may not be necessary to show the menu bar or a document window, and reading the desk scrap into memory is definitely not required. After successfully printing a document, the application should set the file type in the Finder information to Ø. Upon return from the application, the Finder will start up other applications as

necessary to print any remaining files whose type was not set to Ø.
*** The Finder doesn't currently do this, but it may in the future.
***

GLOSSARY

bundle: A resource that maps local IDs of resources to their actual resource IDs; used to provide mappings for file references and icon lists needed by the Finder.

Desktop file: A resource file in which the Finder stores folder resources and the version data, bundle, icons, and file references for each application on the volume.

file reference: A resource that provides the Finder with file and icon information about an application.

file type: A four-character sequence, specified when a file is created, the identifies the type of file.

icon list: A resource consisting of a list of icons.

local ID: A number that refers to an icon list or file reference in an application's resource file and is mapped to an actual resource ID by a bundle.

signature: A four-character sequence that uniquely identifies an application to the Finder.

version data: In an application's resource file, a resource that has the application's signature as its resource type; typically a string that gives the name, version number, and date of the application.

COMMENTS?

Macintosh User Education encourages your comments on this manual.

- What do you like or dislike about it?

- Were you able to find the information you needed?

- Was it complete and accurate?

- Do you have any suggestions for improvement?

Please send your comments to the author (indicated on the cover page) at 10460 Bandley Drive M/S 3-G, Cupertino CA 95014. Mark up a copy of the manual or note your remarks separately. (We'll return your marked-up copy if you like.)

Thanks for your help!

MACINTOSH USER EDUCATION

Putting Together a Macintosh Application /PUTTING/TOGETHER

See Also: Workshop User's Guide for the Lisa
Macintosh Owner's Guide
Inside Macintosh: A Road Map
The Resource Manager: A Programmer's Guide
The Menu Manager: A Programmer's Guide
The Segment Loader: A Programmer's Guide
The Structure of a Macintosh Application

Modification History:			
	First Draft (ROM 2.45)	Caroline Rose	6/9/83
	Second Draft (ROM 4.4)	Caroline Rose	7/14/83
	Third Draft (ROM 7)	Caroline Rose	1/13/84
	Fourth Draft	Caroline Rose	4/9/84
	Fifth Draft	Caroline Rose	7/10/84

ABSTRACT

This manual discusses the fundamentals of preparing, compiling or assembling, and linking a Macintosh application program on the Lisa Workshop development system.

Summary of significant changes and additions since last draft:

- Additions have been made to the interface files and the files you link with or include in your assembly-language source.
- The Resource Compiler has several new features: it lets you designate a nonstandard type of menu; it lets you specify any character by its ASCII code; and it recognizes the types ICN#, FREF, and BNDL, as well as a new general type, GNRL. (See page 7.)
- The default type assumed by MacCom's Lisa->Mac command has changed; the sample Exec file has been changed accordingly (page 14).

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual discusses the fundamentals of preparing, compiling or assembling, and linking a Macintosh application program on the Lisa Workshop development system. It assumes the following:

- You know how to write a Macintosh application in Pascal or assembly language. Details on this may be found in the technical documentation; see Inside Macintosh: A Road Map for a list of all such documentation.

- You're familiar with the Macintosh Finder, which is described in Macintosh, the owner's guide.

- You have a Lisa 2/5 or 2/10 with a Workshop development system (version 2.0), and the Workshop User's Guide for the Lisa. You should also have the Workshop supplement for Macintosh developers, which consists of additional software and documentation that you'll need.

After explaining some conventions it uses, this manual begins by presenting the first steps you should take once your Lisa has been set up for Macintosh application development under the Workshop. It then discusses each of the three files you'll create to develop your application: the source file, the Resource Compiler input file, and an exec file.

The next section discusses how to divide an application into segments. Two utility programs on the Macintosh are then described, followed by important information for programmers who want to write all or part of an application in assembly language.

Finally, there's a summary of the steps to take to put together a Macintosh application.

(note)

This manual presents a recommended scenario, not by any means the only possible one. Details, such as what you name your files, may vary.

Conventions

Sometimes this manual shows you what to do in a two-column table, the first one labeled "Prompt" and the second "Response". The first column shows what appears on the Lisa to "prompt" you; it might be a request for a file name, or just the Workshop command line. This column will not show all the output you'll get from a program, only the line that prompts you. (There may have been a lot of output before that line.) The second column shows what you type as a response. The following notation is used:

Notation	Meaning
<ret>	Press the RETURN key.
[ ]	Explanatory comments are enclosed in [ ]; you don't type them.

A space preceding <ret> is not to be typed. It's there only for readability.

[ ] in the "Prompt" column actually appear in the prompt; they enclose defaults.

Except where indicated otherwise, you may type letters in any combination of uppercase and lowercase, regardless of how they're shown in this manual.

GETTING STARTED

Once your Lisa has been set up for Macintosh application development, it's a good idea to orient yourself to the files installed on it. You can use the List command in the File Manager to list all the file names. Certain subsets of related files begin with the same few letters followed by a slash; some typical naming conventions are as follows:

Beginning of file name	Description
Intrfc/	Text files containing the Pascal interfaces
TlAsm/	Text files to include when using assembly language
Obj/	Object files
Work/	Your current working files
Back/	Backup copies of your working files
Example/	Examples provided by Macintosh Technical Support

(note)
This manual assumes that your files observe the above naming conventions.

You'll write your application to a Macintosh system disk, which means a Macintosh disk that contains the system files needed for running an application. The necessary system files are on the MacStuff 1 disk that you received as part of the Workshop supplement. Use that disk only to create other system disks. Here's how:

1. Insert the MacStuff 1 disk into the Macintosh and open MacStuff 1.

2. Copy the System Folder to a new Macintosh disk; the exact method you use depends on whether you have an external drive. See the Macintosh owner's guide for more information.

(note)
One of the files in the System Folder, Imagewriter, is needed only if you're going to print to an Imagewriter

printer; to save space, you might not want to copy it if you don't need it.

If you also need or want any of the other files on the MacStuff 1 disk, or any of the files on MacStuff 2 or 3, copy them as well. Two of these other files, the Resource Mover and Set File utilities in the Tools folder on MacStuff 1, are described later in this manual.

As described in detail in the following sections, you'll create a source file, Resource Compiler input file, and exec file for your application, insert your Macintosh system disk into the Lisa, and run the exec file. The exec file will compile the source file, link the resulting object file with other required object files, run the Resource Compiler to create the application's resource file, and run a program called MacCom to write the application to the Macintosh disk. When MacCom is done, it will eject the disk; to try out your application, you'll insert the ejected disk into the Macintosh and just open the application's icon.

THE SOURCE FILE

Your working files will of course include the source file for your application. Suppose, for example, that you have an application named Samp. The source file would be Work/Samp.Text and would have the structure shown below.

(note)
"Samp" is used as the application name in all examples in this manual. You don't have to use the exact name of your application; any abbreviation will do.

```
PROGRAM Samp;

{ Samp -- A sample application written in Pascal }
{         by Macintosh User Education  7/2/84    }

[ List the following in the order shown. ]

USES {$U Obj/MemTypes    } MemTypes,
     {$U Obj/QuickDraw   } QuickDraw,
     {$U Obj/OSIntf      } OSIntf,
     {$U Obj/ToolIntf    } ToolIntf,
     {$U Obj/MacPrint    } MacPrint,    [ OPTIONAL ]
     {$U Obj/SANE        } SANE,        [ OPTIONAL ]
     {$U Obj/Elems       } Elems,       [ OPTIONAL ]
     {$U Obj/PackIntf    } PackIntf;    [ OPTIONAL ]

[ Your LABEL, CONST, TYPE, and VAR declarations will be here. ]

[ Your application's procedures and functions will be here. ]

BEGIN

  [ The main program will be here. ]

END.
```

Each line in the USES clause specifies first a file name and then a unit name (which happen to be the same in all cases here). The file contains the compiled Pascal interface for that unit; the corresponding text file name begins with "Intrfc/" rather than "Obj/". The Pascal interface includes the declarations of all the routines in the unit. It also contains any data types, predefined constants, and, in the case of QuickDraw, Pascal global variables.

File name	Interface it contains
Intrfc/MemTypes.Text	Basic Memory Manager data types
Intrfc/QuickDraw.Text	QuickDraw
Intrfc/OSIntf.Text	Operating System
Intrfc/ToolIntf.Text	Toolbox, except QuickDraw
Intrfc/MacPrint.Text	Printing Manager
Intrfc/SANE.Text	Floating-Point Arithmetic Package
Intrfc/Elems.Text	Transcendental Functions Package
Intrfc/PackIntf.Text	Other packages

You only have to include the files for the units your application uses. It doesn't do any harm to include them all, but it will take somewhat longer for your program to compile. If you're using any units of your own, just add their Pascal interface files at the end of the USES clause.

As described in the Segment Loader manual, you can divide the code of an application into several segments and have only some of them in memory at a time. The section "Dividing Your Application Into Segments" tells how to specify segments in your source file. If you don't specify any, your program will consist of a single segment (whose name is blank).

THE RESOURCE COMPILER INPUT FILE

You'll need to create a resource file for your application. This is done with the Resource Compiler, and you'll have among your working files an input file to the Resource Compiler. *** In the future, you'll use the Resource Editor, which doesn't yet exist. *** One convention for naming this input file is to give it the name of your source file followed by "R" (such as Work/SampR.Text).

The first entry in the input file specifies the name to be given to the output file from the Resource Compiler, the resource file itself; you'll enter "Work/" followed by the application name and ".Rsrc". Another entry tells which file the application code segments are to be read from. (As discussed in the Resource Manager manual, the code segments are actually resources of the application.) You'll enter the name of the Linker output file specified in the exec file for building your application, as described in the next section.

If you don't want to include any resources other than the code segments, you can have a simple input file like this:

```
*  SampR -- Resource input for sample application
*           Written by Macintosh User Education  7/2/84

Work/Samp.Rsrc

Type SAMP = STR
  ,Ø
Samp Version 1.Ø -- July 2, 1984

Type CODE
  Work/SampL,Ø
```

This tells the Resource Compiler to write the resulting resource file to Work/Samp.Rsrc and to read the application code segments from Work/SampL.Obj. It also specifies the file's signature and version data, which the Finder needs.

It's a good idea to begin the input file with a comment that describes its contents and shows its author, creation date, and other such information. Any line beginning with an asterisk (*) is treated as a comment and ignored. (You cannot have comments embedded within lines.) The Resource Compiler also ignores the following:

- leading spaces (except before the text of a string resource)
- embedded spaces (except in file names, titles, or other text strings)
- blank lines (except for those indicated as required)

The first line that isn't ignored specifies the name to be given to the resulting resource file. Then, for each type of resource to be defined, there are one or more "Type statements". A Type statement consists of the word "Type" followed by the resource type (without quotes) and, below that, an entry of following format for each resource:

```
file name!resource name,resource ID (resource attributes)
type-specific data
```

The punctuation shown here in the first line is typed as part of the format. You must always provide a resource ID. Specifications other than the resource ID may or may not be required, depending on the resource type:

- Either there will be some type-specific data defining the resource or you'll give a file name indicating where the resource will be read from. Even in the absence of a file name, you **must** include the comma before the resource ID.

- You specify a resource name along with the file name for fonts and drivers. The Menu Manager procedures AddResMenu and InsertResMenu will put these resource names in menus. Enter the names in the combination of uppercase and lowercase that you want to appear in the menus.

- Resource attributes in parentheses are optional for all types. They're given as a number equal to the value of the resource attributes byte, and Ø is assumed if none is specified. (See the Resource Manager manual for details.) For example, for a resource that's purgeable but has no other attributes set, the input will be "(32)".

If you want to enter a nonprinting or other unusual character in your input file, either by itself or embedded within text, just type a back slash (\) followed by the ASCII code of the character in hexadecimal. For example, the Resource Compiler interprets \ØD as a Return character and \14 as the apple symbol.

The formats for the different types of resources are best explained by example. Some examples are given below along with remarks that provide further explanation. Here are some points to remember:

- Most examples list only one resource per Type statement, but you can include as many resources as you like in a single statement.

- In every case, resource attributes in parentheses may be specified after the resource ID.

- All numbers are base 1Ø except where hexadecimal is indicated.

- The Type statements may appear in any order in the input file.

```
Type WIND                       Window template
  ,128                          Resource ID
  Status Report                 Window title
  4Ø 8Ø 12Ø 3ØØ                 BoundsRect (top left bottom right)
  Visible GoAway                For FALSE, use Invisible or NoGoAway
  Ø                             ProcID (window definition ID)
  Ø                             RefCon (reference value)

Type MENU                       Menu, standard type
  ,128                          Resource ID (becomes the menu ID)
* menu for desk accessories
  \14                           Menu title (apple symbol)
    About Samp...               Menu item
                                Blank line required at end of menu
  ,129                          Resource ID
  Edit                          Menu title
    Cut/X                       Menu items, one per line, with meta-
    Paste/Z                      characters, ! alone for check mark
    (-                          You cannot specify a blank item; use (-
    Word Wrap!                   for a disabled continuous line.
                                Blank line required at end of menu
```

```
Type MENU                       Menu, nonstandard type
  ,2ØØ                          Resource ID   [ SEE NOTE 1 BELOW ]
  2Ø1                           Resource ID of menu definition procedure
  Patterns                      Menu title (may be followed by items)
                                Blank line required at end of menu
Type CNTL                       Control template
  ,128                          Resource ID
  Help                          Control title
  55 2Ø 75 9Ø                   BoundsRect
  Visible                       For FALSE, use Invisible
  Ø                             ProcID (control definition ID)
  1                             RefCon (reference value)
  Ø Ø Ø                         Value minimum maximum

Type ALRT                       Alert template
  ,128                          Resource ID
  12Ø 1ØØ 19Ø 25Ø               BoundsRect
  3ØØ                           Resource ID of item list
  F721                          Stages word in hexadecimal

Type DLOG                       Dialog template
  ,128                          Resource ID
* modal dialog
  1ØØ 1ØØ 19Ø 25Ø               BoundsRect
  Visible 1 NoGoAway Ø          1 is procID, Ø is refCon
  2ØØ                           Resource ID of item list
                                Title (none in this case)
  ,129
* modeless dialog
  1ØØ 1ØØ 19Ø 25Ø               BoundsRect
  Visible Ø GoAway Ø            Ø procID, Ø refCon
  3ØØ                           Resource ID of item list
  Find and Replace              Title

Type DITL                       Item list in dialog or alert
  ,2ØØ                          Resource ID
  5                             Number of items
  BtnItem Enabled               Also:  ChkItem, RadioItem
  6Ø 1Ø 8Ø 7Ø                   Display rectangle
  Start                         Title
                                Blank line required between items
  ResCItem Enabled              Control defined in control template
  6Ø 3Ø 8Ø 1ØØ                  Display rectangle
  128                           Resource ID of control template

  StatText Disabled             Also:  EditText
  1Ø 93 26 13Ø                  Display rectangle
  Seed                          The text (may be blank if EditText)

  IconItem Disabled             Also:  PicItem
  1Ø 24 42 56                   Display rectangle
  128                           Resource ID of icon
```

```
  UserItem Disabled          Application-defined item
  2Ø 5Ø 6Ø 85                Display rectangle

Type ICON                    Icon
  ,128                       Resource ID
  Ø38Ø ØØØØ                  The icon in hexadecimal (32 such lines
    . . .                     altogether)
  1ECØ 318Ø

Type ICN#                    Icon list
  ,128                       Resource ID
  2                          Number of icons
  ØØØ1 ØØØØ                  The icons in hexadecimal (32 such lines
    . . .                     altogether for each icon)
  ØØØ2 8ØØØ

Type CURS                    Cursor
  ,3ØØ                       Resource ID
  7FFC . . . 287F            The data:  64 hex digits on one line
  ØFCØ . . . 1FF8            The mask:  64 hex digits on one line
  ØØØ8 ØØØ8                  The hotSpot in hexadecimal (v h)

Type PAT                     Pattern
  ,2ØØ                       Resource ID
  AADDAA66AADDAA66           The pattern in hexadecimal

Type PAT#                    Pattern list
  ,136                       Resource ID
  2                          Number of patterns
  5522552255225522           The patterns in hexadecimal, one per
  FFEEDDCCFFEEDDCC            line

Type STR                     String
  ,128                       Resource ID
This is your string          The string on one line (leading spaces
                              not ignored)
Type STR#                    String list
  ,129                       Resource ID
First string                 The strings
Second string
* note Return in next string
Third string\ØDcontinued
                             Blank line required after last string
Type DRVR                    Desk accessory or other device driver
  Obj/Monkey!Monkey,17 (32)  File name!resource name,resource ID
                              [ SEE NOTE 2 BELOW ]
Type FREF                    File reference
  ,128                       Resource ID
  APPL  Ø  TgFil             File type  local ID of icon  file name
                              (omit file name if none)
```

```
Type BNDL                       Bundle
  ,128                          Resource ID
  SAMP  Ø                       Bundle owner
  2                             Number of types in bundle
  ICN#  1                       Type and number of resources
  Ø  128                        Local ID Ø maps to resource ID 128
  FREF  1                       Type and number of resources
  Ø  128                        Local ID Ø maps to resource ID 128

Type FONT                       Font (or FWID for font widths)
  Obj/Griffin!Griffin,4ØØ@Ø     File name!resource name,resource ID
  Obj/Griffin1Ø,4ØØ@1Ø          File name,resource ID   [ SEE NOTE 3 ]
  Obj/Griffin12,4ØØ@12          File name,resource ID   [ BELOW      ]

Type CODE                       Application code segments
  Obj/SampL,Ø                   Linker output file name,resource ID
                                 [ SEE NOTE 4 BELOW ]
```

Notes:

1. Notice that the input for a nonstandard menu has one extra line in it: the resource ID of the menu definition procedure, just following the resource ID of the menu. If that line is omitted (that is, if the menu's resource ID is followed by a line containing text rather than a number), the resource ID of the standard menu definition procedure (Ø) is assumed.

2. The Resource Compiler adds a NUL character (ASCII code Ø) at the beginning of the name you specify for a 'DRVR' type of resource. This inclusion of a nonprinting character avoids conflict with file names that are the same as the names of desk accessories.

3. The resource ID for a font resource has a special format:

   font number @ size

   The actual resource ID that the Resource Compiler assigns to the font is

   (128 * font number) + size

   Three font resources are listed in the example above. Size Ø is used to provide only the name of the font (Griffin in this case); a file name must also be specified but is ignored. The two remaining font resources define the Griffin font in two sizes, 1Ø and 12.

4. For a 'CODE' type of resource, ".Obj" is appended to the given file name, and the resource ID you specify is ignored. The Resource Compiler always creates two resources of this type, with ID numbers Ø and 1, and will create additional ones numbered sequentially from 2 if your program is divided into segments.

The Type statement for a resource of type 'WDEF', 'MDEF', 'CDEF', 'FKEY' *** function key code ***, 'KEYC', 'PACK', or 'PICT' has the same format as for 'CODE': only a file name and a resource ID are specified. For the 'PICT' type, the file contains the picture; for the other types, it contains the compiled code of the resource, and the Resource Compiler appends ".Obj" to the file name.

(note)
The 'MBAR' resource type is not recognized by the Resource Compiler.

If your application is going to write to the resulting resource file as well as read it, you should place the Type statement for the code segments at the end of the input file. In general, any resources that the application might change and write out to the resource file should be listed first in the input file, and any resources that won't be changed (like the code segments) should be listed last. The reason for this is that the Resource Compiler stores resources in the reverse of the order that they're listed, and it's more efficient for the Resource Manager to do file compaction if the changed resources are at the end of the resource file.

Defining Your Own Resource Types

You can use one of the three types GNRL, HEXA, and ANYB to define your own types of resources in the Resource Compiler input file. GNRL allows you to specify your resource data in the manner best suited to your particular data format; you specify the data as you want it to appear in the resource. A code (beginning with a period) tells the Resource Compiler how to interpret what you enter on the next line or lines (up to the next code or the end of the Type statement). The following illustrates all the codes:

```
Type GNRL                    General type
  ,128                       Resource ID
  .P                         Pascal strings (with length byte), one
A Pascal string               per line
Another Pascal string
  .S                         Strings without length byte, one per
A string                      line
  .I                         Integers (decimal), one per line
  Ø
  1
  .L                         Long integers (decimal), one per line
  5438
  .H                         Bytes in hexadecimal, any number
  526FEEC942E78EA4            total, any number per line
  ØF4C
  .B                         Bytes from a file
  MyData  36  256            File name  number of bytes  offset
                             Blank line required at end of statement
```

You can use an equal sign (=) along with the GNRL type to define a

resource of any desired format and with any four-character resource type; for example, to define a resource of type 'MINE' consisting of the integer 57 followed by the Pascal string 'Finance charges', you could enter this:

```
Type MINE = GNRL
  ,4ØØ
  .I
  57
  .P
Finance charges
```

The Resource Manager call GetResource('MINE',4ØØ) would return a handle to this resource.

The types HEXA and ANYB simply offer alternatives to the .H and .B options (respectively) of the GNRL type, as shown below.

```
Type HEXA                    Bytes in hexadecimal
  ,2Ø1                       Resource ID
  526FEEC942E78EA4           The bytes (any number total, any
  ØF4C                        number per line)
                             Blank line required at end
Type ANYB                    Bytes from a file
  MyData,2ØØ                 File name,resource ID
  36  256                    Number of bytes  offset in file
```

You can also define a new resource type that inherits the properties of a standard type. For example,

```
Type XDEF = WDEF
```

defines the new type 'XDEF', which the Resource Compiler treats exactly like 'WDEF'. The next line would contain a file name and resource ID just as for a 'WDEF' resource.

THE EXEC FILE

It's useful for each application to have an exec file that does everything necessary to build the application, including compiling, linking, creating the resource file, and writing to a Macintosh disk. The name of the exec file might, for example, be the source file name followed by "X" (for "eXec"). Work/SampX.Text, the exec file for the Samp application, is shown below.

```
$EXEC
P{ascal}Work/Samp
{no list file}
{default I-code file}
G{enerate}Work/Samp
{default output file}
L{ink}?
+X
{no more options}
Work/Samp
Obj/QuickDraw
Obj/OSTraps
Obj/ToolTraps
Obj/PrLink          [ OPTIONAL ]
Obj/PrScreen        [ OPTIONAL ]
Obj/ElemsAsm        [ OPTIONAL ]
Obj/SANE            [ OPTIONAL ]
Obj/SANEAsm         [ OPTIONAL ]
Obj/PackTraps       [ OPTIONAL ]
Obj/MacPasLib
{end of input files}
{listing to console}
Work/SampL
R{un}RMaker
Work/SampR
R{un}MacCom
F{inder info}Y{es}L{isa->Mac}Work/Samp.Rsrc
Samp
APPL
SAMP
{no bundle bit}
E{ject}Q{uit}
$ENDEXEC
```

The file begins with $EXEC and ends with $ENDEXEC. Everything in between (except for comments in braces) is exactly what you would type on your Lisa if you were not using an exec file. To show what the various entries in this file accomplish, the table below indicates what each of them is a response to, and shows your response as it is in the exec file or as it would be if you were using the keyboard. The numbers on the left are given for reference in the explanation that follows the table.

	Prompt	Response
[1]	Workshop command line	P [for Pascal]
	Input file - [.TEXT]	Work/Samp <ret>
	List file - [.TEXT]	<ret> [for none]
	I-code file - [Work/Samp][.I]	<ret> [for Work/Samp.I]
[2]	Workshop command line	G [for Generate]
	Input file - [.I]	Work/Samp <ret>
	Output file - [Work/Samp][.OBJ]	<ret> [for Work/Samp.Obj]

[3]	Workshop command line	L [for Link]
	Input file [.OBJ] ?	? <ret> [for options]
	Options ?	+X <ret>
	Options ?	<ret> [no more options]
	Input file [.OBJ] ?	Work/Samp <ret>
	Input file [.OBJ] ?	Obj/QuickDraw <ret>
	Input file [.OBJ] ?	Obj/OSTraps <ret>
	. . .	[other input files]
	Input file [.OBJ] ?	Obj/MacPasLib <ret>
	Input file [.OBJ] ?	<ret> [end of input files]
	Listing file [-CONSOLE] / [.TEXT]	<ret> [for -CONSOLE]
	Output file ? [OBJ.]	Work/SampL <ret>
[4]	Workshop command line	R [for Run]
	Run what program?	RMaker <ret>
	Input file [sysResDef][.TEXT] -	Work/SampR <ret>
[5]	Workshop command line	R [for Run]
	Run what program?	MacCom <ret>
	MacCom command line	F [for Finder info]
	Always set Finder info yourself when writing a Mac file? (Y or N)	Y [for Yes]
	MacCom command line	L [for Lisa->Mac]
	Lisa files to write to Mac disk?	Work/Samp.Rsrc <ret>
	Copy to what Mac file?	Samp <ret>
	Type? [????]	APPL <ret>
	Creator? [????]	SAMP <ret>
	Set the Bundle Bit? (Y or N) [No]	<ret> [for No]
	MacCom command line	E [for Eject]
	MacCom command line	Q [for Quit]

Here's what you accomplish at each of the steps:

1. You perform the first part of the compilation process, translating the Pascal source code (Work/Samp.Text) into I-code (Work/Samp.I).

2. You perform the second part of the compilation process, generating an object file (Work/Samp.Obj) from the I-code.

3. You link the application's object file with other object files (resulting in the output file Work/SampL.Obj).

4. You run the Resource Compiler to create the application's resource file (Work/Samp.Rsrc, as specified in Work/SampR.Text, the input file to the Resource Compiler). Included in the resources are the application's code segments, which are read from the Linker output file.

5. You use the MacCom program to write the resource file to the Macintosh disk, giving the file the exact name you want your application to have. You set its file type to 'APPL' and its creator to the signature specified in the resource file. Since there's no bundle in Samp's resource file, you don't set the bundle bit. (See The Structure of a Macintosh Application for more information.) Finally, you ask MacCom to eject the disk.

The files linked with the application's object file in step 3 are described below. Most of them contain a trap interface, which is a set of small assembly-language routines that make it possible to call the corresponding unit or units from Pascal. The files should be listed in the order shown. Specify the optional files only if your application uses the routines they apply to.

File name	Description
Obj/MemTypes.Obj	Basic Memory Manager data types
Obj/QuickDraw.Obj	Pascal interface to QuickDraw, needed so the Linker will know how many QuickDraw globals there are
Obj/OSTraps.Obj	Trap interface for the Operating System
Obj/ToolTraps.Obj	Trap interface for the Toolbox (except QuickDraw)
Obj/PrLink.Obj	Trap interface for the Printing Manager
Obj/PrScreen.Obj	Trap interface for low-level printing routines
Obj/ElemsAsm.Obj	Trap interface for the Transcendental Functions Package
Obj/SANE.Obj	First part of the trap interface for the Floating-Point Arithmetic Package
Obj/SANEAsm.Obj	Second part of the trap interface for the Floating-Point Arithmetic Package
Obj/PackTraps.Obj	Trap interface for other packages
Obj/MacPasLib.Obj	Intrinsic Pascal routines, such as Mod and Concat

Before running the Exec file, insert a Macintosh system disk into the Lisa. Run the exec file as follows:

Prompt	Response
Workshop command line	R [for Run]
Run what program?	<Work/SampX <ret>

When the disk is ejected, remove it and insert it into the Macintosh. To try out your application, just open its icon.

(warning)

If you don't set your application's file type and creator, either you won't be able to open its icon in the usual way, or a different application may start up when you do open it!

Notice that if you change the application's signature or the setting of its bundle bit, step 5 of the above exec file will have to be edited accordingly. Furthermore, if you modify one of the resources related to the Finder interface and the change doesn't seem to have taken effect, try holding down the Option and Command keys when you start up the system disk on the Macintosh. This is necessary, for example, if you change only the application's icon.

(note)

An unfortunate side effect of pressing Option-Command during startup is that all folders will be lost--but they're easy enough to recreate, and you shouldn't have to do this too often.

Before making major changes to your application, it's a good idea to back it up. You can use the Backup command in the File Manager to back up all files beginning with "Work/" to files beginning with "Back/" (Work/=,Back/=). Also, you might want to periodically back up your working files onto 3 1/2-inch disks.

There are several ways you could refine the exec file illustrated here; exactly what you do will depend on your particular situation. Some possibilities are listed below.

- You can set up the exec file to compile or link only if actually necessary. For more information, see your Workshop documentation or the sample general-purpose exec file provided in the Workshop supplement.

- To save disk space, you can add commands to the exec file to make it delete the three intermediate files: the I-code and object files for the application and the Linker output file.

- If you want to keep the intermediate files around but are working on more than one application, you can save disk space by giving the intermediate files the same name for all applications (say, "Work/Temp").

- You can embed the exec file in your program's source file. To do this, you must use "(*" and "*)" around the exec part of the file and use the I invocation option. See your Workshop documentation for details.

DIVIDING YOUR APPLICATION INTO SEGMENTS

You can specify the beginning of a segment in your application's source file as follows:

```
{$S segname}
```

where segname is the segment name, a sequence of up to eight characters. Normally you should give the main segment a blank name. For example, you might structure your program as follows:

```
PROGRAM Samp;

[ The USES clause and your LABEL, CONST, and VAR declarations
  will be here. ]

{$S Seg1}

[ The procedures and functions in Seg1 will be here. ]

{$S Seg2}

[ The procedures and functions in Seg2 will be here. ]

{$S     }

BEGIN

  [ The main program will be here. ]

END.
```

You can specify the same segment name more than once; the routines will just be accumulated into that segment. To avoid problems when moving routines around in the source file, some programmers follow the practice of putting a segment name specification before every routine.

(warning)
Uppercase and lowercase letters **are** distinguished in segment names. For example, "Seg1" and "SEG1" are not equivalent names.

If you don't specify a segment name before the first routine in your file, the blank segment name will be assumed there.

You can also specify what segment the routines in a particular file should be in by using the Lisa utility program ChangeSeg. For example, suppose you want to give your main segment a nonblank name (say, "SegMain"); you can't do this without using ChangeSeg, because the Linker puts the intrinsic Pascal routines in the blank-named segment, and they must be in the same segment as your main program. You can use ChangeSeg as shown below to tell the Linker to put the intrinsic Pascal routines, which are in Obj/MacPasLib, in the segment named SegMain.

Prompt	Response
Workshop command line	R [for Run]
Run what program?	ChangeSeg <ret>
File to change:	Obj/MacPasLib <ret>
Map all Names ? (Y/N)	Y [for Yes]
New Seg name ?	SegMain <ret>

(note)
This changes the segment name for **all** programs you might be developing.

WORKING WITH RESOURCE FILES ON THE MACINTOSH

The Resource Mover utility (RMover for short) lets you examine and manipulate resources on the Macintosh. It's especially useful for working with the system resource file, such as to see what resources it contains. You can work with any of the resource files on the disk that's in the Macintosh. Resources can be copied, moved, or removed, and their attributes can be changed. RMover also enables you to store a QuickDraw picture in a resource file.

To get a copy of RMover on a Macintosh system disk, just copy it from the Tools folder on the MacStuff 1 disk.

To use RMover, open its icon. You'll see a window whose title shows the name of the disk RMover is on. Inside the window will be a list of the names and sizes of all the resource files on the disk. To look at one of the resource files, select it by clicking its entry in the list and then choosing Open from the File menu. A window will appear that contains a list of "Type" entries representing the resources in that file. For each resource type in the file, there's a bold entry consisting of the word "**Type**" followed by the resource type. Under the bold entry, there's an indented entry for each resource of that type, showing the resource ID and the resource name, if any, for that resource.

To select a particular resource that you want to examine or manipulate, just click its entry in the resource file window. You can select several successive resources by dragging through their entries, or extend or shorten a selection by pressing Shift while clicking or dragging. To select every resource of a particular type, click the bold entry for that type.

If you want more information about the selected resource or resources, choose Open from the File menu. For each resource, a window will appear that also shows the resource's size, attributes that you can change by clicking them (such as purgeable and protected), and, where possible, a depiction of the resource itself. For example, an icon or picture resource will be displayed in the resource window; for a font resource, a line of text will appear in that font and size and tell the name and size of the font.

Use the Cut, Copy, Paste, and Clear commands in RMover's Edit menu to manipulate resources. They perform the same functions as in text editing except that they apply to the resource itself. For example, Cut will remove a selected resource from the resource file and place it on the Clipboard *** (currently called the Scrap in RMover) ***. You can then activate another resource file window and use Paste to copy the resource on the Clipboard into that file. If there's already a resource with the ID number of the resource being pasted, it's replaced; otherwise, the resource being pasted is added to the file.

(note)

You can't use RMover to paste something that you cut or copied in a previous application. You can only paste resources cut or copied within RMover itself.

The Set ID and Set Name commands in the Edit menu let you change a resource's ID number or name. They operate on the selected resource or, if more than one is selected, on the first selected resource.

You can get a QuickDraw picture into a resource file as follows: draw the picture in MacPaint; cut or copy it, to put it on the Clipboard; open the Scrapbook desk accessory; copy the picture into the Scrapbook by choosing Paste from the Edit menu; return to the Finder and use RMover to move or copy the picture from the Scrapbook resource file into your resource file.

SETTING FILE INFORMATION ON THE MACINTOSH

Normally you'll set your application's file type and creator from the MacCom program on the Lisa, as described earlier. You don't have to set this information for the files you copy from the MacStuff disks, because it gets copied along with them. Still, situations may arise that require setting the file type and creator on the Macintosh rather than the Lisa; if so, you can do this with the Set File utility on the MacStuff 1 disk. Set File is also useful for examining the types and creators of existing files that you've received.

Set File displays a list of the file names on your disk (including DeskTop, which you can't see on the desktop and should just ignore). Select a file name from this list by clicking it. Enter the file type in the Type box and the creator in the Creator box. Click the Bundle check box if you want the bundle bit set. Click "Set it" to make your settings take effect. Do this for as many files as you like; when you're done, click "Quit".

*** To type an entry in the Type or Creator box, you should always press the Tab key to select the current contents of the box and then replace it with what you type. Don't just click or drag to select in the box; due to a temporary quirk in Set File, this may not work. ***

(warning)
Remember, uppercase and lowercase characters **are** distinguished in what you enter for file types and creators.

If you don't know the creator and type for a particular application file, you can just set the type to APPL and leave the Creator box blank; the Finder will use its default icon for application programs (rather than the program's distinctive icon, if any) but will still let you open the icon as usual.

If you ever get into a situation where Set File's own file type and creator aren't set (preventing you from opening it in the usual way), you can "trick" the Finder by holding down the Option and Command keys while double-clicking Set File's icon; this bypasses the Finder's requirement that the file type and creator be set.

(warning)
Option-Command-double-clicking an icon that doesn't belong to an application will cause a fatal system error.

NOTES FOR ASSEMBLY-LANGUAGE PROGRAMMERS

You can write all or part of your Macintosh application in assembly language. Suppose, for example, that you write most of it in Pascal but have some utility routines written in assembly language. Your working files will include a source file and object file for the assembly-language routines (say, Work/SampA.Text and Work/SampA.Obj). The source file will have the structure shown below.

```
; SampA -- Assembly-language routines for Samp
;          Written by Macintosh User Education  7/2/84

[ List the following in the order shown. ]

.INCLUDE TlAsm/SysEqu.Text
.INCLUDE TlAsm/SysMacs.Text
.INCLUDE TlAsm/SysErr.Text
.INCLUDE TlAsm/GrafEqu.Text
.INCLUDE TlAsm/GrafTypes.Text
.INCLUDE TlAsm/QuickMacs.Text
.INCLUDE TlAsm/ToolMacs.Text
.INCLUDE TlAsm/ToolEqu.Text
.INCLUDE TlAsm/ResEqu.Text
.INCLUDE TlAsm/PrEqu.Text        [ OPTIONAL ]
.INCLUDE TlAsm/SANEMacs.Text     [ OPTIONAL ]
.INCLUDE TlAsm/PackMacs.Text     [ OPTIONAL ]
.INCLUDE TlAsm/PackEqu.Text      [ OPTIONAL ]
.INCLUDE TlAsm/HeapDefs.Text     [ OPTIONAL ]
.INCLUDE TlAsm/FSEqu.Text        [ OPTIONAL ]

[ Here there will be a .PROC or .FUNC directive for each routine, ]
[ followed by the routine itself.  Two examples follow.  ]

; PROCEDURE MyRoutine (count: INTEGER);

.PROC MyRoutine

MyRoutine
      [ the code of MyRoutine ]

; FUNCTION MyOtherRoutine : LongInt;

.FUNC MyOtherRoutine

MyOtherRoutine
      [ the code of MyOtherRoutine ]

.END
```

(note)

The .PROC or .FUNC directive clears the symbol table, so symbols defined in one routine can't be referred to in another (without an explicit reference using .REF). If

you want to share code between routines, you can instead have a single .PROC directive for SampA followed by a .DEF directive for each routine name.

Including unneeded files with .INCLUDE directives will do no harm except make your program take longer to assemble. The files marked as optional above are the least commonly needed; even some of the others may not be required. Here's what the files contain:

File name	Description
TlAsm/SysEqu.Text	System equates (memory layout and system data structures)
TlAsm/SysMacs.Text	System macros
TlAsm/SysErr.Text	System error equates
TlAsm/GrafEqu.Text	System graphics equates (cursor-related)
TlAsm/GrafTypes.Text	QuickDraw equates (constants, offsets to locations of global variables, and offsets into structures)
TlAsm/QuickMacs.Text	QuickDraw macros
TlAsm/ToolMacs.Text	Toolbox macros, except QuickDraw
TlAsm/ToolEqu.Text	Toolbox equates, except QuickDraw (constants, locations of system globals, and offsets into structures)
TlAsm/ResEqu.Text	Equates for resources (resource types, standard resource IDs)
TlAsm/PrEqu.Text	Equates for Printing Manager
TlAsm/SANEMacs.Text	Macros and equates for Floating-Point Arithmetic and Transcendental Functions Packages
TlAsm/PackMacs.Text	Macros for other packages
TlAsm/PackEqu.Text	Equates for other packages
TlAsm/HeapDefs.Text	Memory Manager equates
TlAsm/FSEqu.Text	File system equates

If you've created any similar files for units of your own, just add .INCLUDE directives for them after the last .INCLUDE directive shown above.

To specify the beginning of a segment in assembly language, you can use the directive

```
.SEG 'segname'
```

where segname is the segment name, a sequence of up to eight characters.

For each assembly-language routine invoked from Pascal, the Pascal source file for your application will include an external declaration. For example:

```
PROCEDURE MyRoutine (count: INTEGER); EXTERNAL;
FUNCTION  MyOtherRoutine : LongInt; EXTERNAL;
```

If the routines form a unit that may be used by other applications, you should instead prepare a Pascal interface file for the unit and include it in the USES clause in the application's source file.

You'll assemble the Work/SampA.Text file as shown below.

Prompt	Response
Workshop command line	A [for Assemble]
Input file - [.TEXT]	Work/SampA <ret>
Listing file (<CR> for none) - [.TEXT]	<ret> [for none]
Output file - [Work/SampA] [.OBJ]	<ret> [for Work/SampA.Obj]

(note)
If you do want a listing file, you may want to put a .NOLIST directive before your first .INCLUDE and a .LIST after your last one, so the contents of all the included files won't appear in the listing.

You can assemble the code manually and then, after you've created or changed the Pascal source file, use the exec file for the application as illustrated earlier (adding the name of the assembly-language object file to the list of Linker input files). You may also want to set up an exec file that just assembles the assembly-language routines and links the resulting object file with everything else, for when you've changed only those routines and not the Pascal program. This exec file would begin with the responses listed above and then continue with step 3 of the exec file illustrated earlier.

If the entire application is written in assembly language, the source file will have the same structure as the one shown above, but you'll also need to have a "dummy" Pascal program (in Work/Samp.Text):

```
PROGRAM Samp;

{ Samp -- A sample application written in assembly language }
{         by Macintosh User Education  7/2/84               }

PROCEDURE SampA; EXTERNAL;

BEGIN

  SampA

END.
```

If the application has its own globals, you'll need to reserve space for them after QuickDraw's globals. You can do this by adding the following at the beginning of the dummy program:

```
USES {$U Obj/QuickDraw    } QuickDraw;

VAR globalSpace: ARRAY [1..x] OF INTEGER;
```

where x is the number of words (not bytes) occupied by your globals. To know where to access the globals, your assembly-language program must store the address of the globalSpace array somewhere (conventionally at 28(A5), the end of the application parameter area). You could set the program up to receive this address as a parameter; the EXTERNAL declaration in the dummy program would be

```
PROCEDURE SampA (globalPtr: Ptr); EXTERNAL;
```

and the call to SampA would be

```
SampA(@globalSpace[1])
```

Furthermore, you can save about 6K of memory if you're programming entirely in assembly language, by removing Obj/MacPasLib from the list of files to link with and instead including TlAsm/MocPasLib.Text in your source file.

SUMMARY OF PUTTING TOGETHER AN APPLICATION

This summary assumes the file-naming conventions presented in the "Getting Started" section. Page numbers indicate where details may be found.

ONE TIME ONLY:

- Prepare a Macintosh system disk by copying the System Folder from the MacStuff 1 disk to a new Macintosh disk (page 4).

- On the Lisa, use the Editor (via the Edit command) to create the exec file (page 14).

ONCE PER VERSION OF YOUR APPLICATION'S SOURCE/RESOURCES:

- On the Lisa, use the Editor to create or edit the application source file (page 6) or the Resource Compiler input file for your application's resources (page 7).

- Insert the Macintosh system disk into the Lisa.

- On the Lisa, run the exec file (page 17). It will eject the Macintosh disk when done.

- To try out your application, remove the disk from the Lisa, insert it into the Macintosh, and open the application's icon.

- When appropriate, back up your working files by using the Backup command in the File Manager to copy Work/= to Back/=, or onto a 3 1/2-inch disk (with, for example, Backup Work/= to -lower-=).

COMMENTS?

Macintosh User Education encourages your comments on this manual.

- What do you like or dislike about it?

- Were you able to find the information you needed?

- Was it complete and accurate?

- Do you have any suggestions for improvement?

Please send your comments to the author (indicated on the cover page) at 10460 Bandley Drive M/S 3-G, Cupertino CA 95014. Mark up a copy of the manual or note your remarks separately. (We'll return your marked-up copy if you like.)

Thanks for your help!

```
{SX-}
PROGRAM Edit;

  { Edit -- A small sample application written in Pascal }
  {         by Macintosh User Education                  }

   USES {SU-}
      {SU Obj/QuickDraw   } QuickDraw,
      {SU Obj/OSIntf      } OSIntf,
      {SU Obj/ToolIntf    } ToolIntf;

   CONST
      lastMenu = 3; { number of menus }
      appleMenu = 1; { menu ID for desk accessory menu }
      fileMenu = 256; { menu ID for File menu }
      editMenu = 257; { menu ID for Edit menu }

   VAR
      myMenus: ARRAY [1..lastMenu] OF MenuHandle;
      screenRect, dragRect, pRect: Rect;
      doneFlag, temp: BOOLEAN;
      myEvent: EventRecord;
      code, refNum: INTEGER;
      wRecord: WindowRecord;
      myWindow, whichWindow: WindowPtr;
      theMenu, theItem: INTEGER;
      hTE: TEHandle;

   PROCEDURE SetUpMenus;
   { Once-only initialization for menus }

      VAR
         i: INTEGER;
         appleTitle: STRING[1];

      BEGIN
         InitMenus; { initialize Menu Manager }
         appleTitle := ' '; appleTitle[1] := CHR(appleSymbol);
         myMenus[1] := NewMenu(appleMenu, appleTitle);
         AddResMenu(myMenus[1], 'DRVR'); { desk accessories }
         myMenus[2] := GetMenu(fileMenu);
         myMenus[3] := GetMenu(editMenu);
         FOR i := 1 TO lastMenu DO InsertMenu(myMenus[i], 0);
         DrawMenuBar;
      END; { of SetUpMenus }

   PROCEDURE DoCommand(mResult: LongInt);

      VAR
         name: STR255;

      BEGIN
         theMenu := HiWord(mResult); theItem := LoWord(mResult);
         CASE theMenu OF

            appleMenu:
               BEGIN
               GetItem(myMenus[1], theItem, name);
               refNum := OpenDeskAcc(name);
               END;

            fileMenu: doneFlag := TRUE; { Quit }

            editMenu:
               BEGIN
               IF NOT SystemEdit(theItem-1) THEN
                  BEGIN
                  SetPort(myWindow);
```

```
                CASE theItem OF

                   1: TECut(hTE);

                   2: TECopy(hTE);

                   3: TEPaste(hTE);

                END; { of item case }
                END;
             END; { of editMenu }

       END; { of menu case }
       HiliteMenu(0);

    END; { of DoCommand }

BEGIN { main program }
    InitGraf(@thePort);
    InitFonts;
    FlushEvents(everyEvent,0);
    InitWindows;
    SetUpMenus;
    TEInit;
    InitDialogs(NIL);
    InitCursor;

    screenRect := screenBits.bounds;
    SetRect(dragRect,4,24,screenRect.right-4,screenRect.bottom-4);
    doneFlag := FALSE;

    myWindow := GetNewWindow(256,@wRecord,POINTER(-1));
    SetPort(myWindow);

    pRect := thePort^.portRect;
    InsetRect(pRect,4,0);
    hTE := TENew(pRect,pRect);
    REPEAT
       SystemTask;
       TEIdle(hTE);
       temp := GetNextEvent(everyEvent,myEvent);
       CASE myEvent.what OF

          mouseDown:
             BEGIN
             code := FindWindow(myEvent.where,whichWindow);
             CASE code OF

                inMenuBar: DoCommand(MenuSelect(myEvent.where));

                inSysWindow: SystemClick(myEvent,whichWindow);

                inDrag: DragWindow(whichWindow,myEvent.where,dragRect);

                inGrow,inContent:
                   BEGIN
                   IF whichWindow<>FrontWindow THEN
                      SelectWindow(whichWindow)
                   ELSE
                      BEGIN
                      GlobalToLocal(myEvent.where);
                      TEClick(myEvent.where,FALSE,hTE);
                      END;
                   END;

             END; { of code case }
             END; { of mouseDown }
```

```
         keyDown, autoKey:
            IF myWindow=FrontWindow THEN
               TEKey(CHR(myEvent.message MOD 256),hTE);

         activateEvt:
            IF ODD(myEvent.modifiers) { window is becoming active }
               THEN
               TEActivate(hTE)
            ELSE
               TEDeactivate(hTE);

         updateEvt:
            BEGIN
            SetPort(myWindow);
            BeginUpdate(myWindow);
            TEUpdate(thePort^.portRect,hTE);
            EndUpdate(myWindow);
            END; { of updateEvt }

      END; { of event case }

   UNTIL doneFlag;
END.
```

```
*  EditResDef -- Resource input for small sample application
*                Written by Macintosh User Education

Example/Edit.Rsrc

Type MENU
  ,256
  File
    Quit

  ,257
  Edit
    Cut
    Copy
    Paste

Type WIND
  ,256
  A Sample
  50 40 300 450
  Visible NoGoAway
  0
  0

Type EDIT = STR
,0
Edit Version 1.0    - 12 December 83

Type CODE
 Example/editL,0
```

```
{SX-}
{SR-}
PROGRAM Scroll;
{-------------------------------------------------------------------------------
    This is a simple program to demonstrate how to use scroll bars.
    You can scroll text or graphics or both.
    You can scroll horizontally or vertically.
    By Cary Clark, Macintosh Technical Support          Apple Computer Inc., 1984
 -------------------------------------------------------------------------------}

USES
      {SU-}
      {SU Obj/QuickDraw } QuickDraw,
      {SU Obj/OSIntf    } OSIntf,
      {SU Obj/ToolIntf  } ToolIntf;

CONST
      Horizontal = 1;     {These are the choices in the menu 'Scroll Bar'}
      Vertical = 2;
      TextItem = 4;
      Graphics = 5;

      FileMenu = 1;       {Resource numbers and position in the Menu bar}
      ScrollMenu = 2;

      NumOfRects = 30;    {quantity of rectangles and strings to scroll around}
      NumOfStrings = 55;

TYPE
      MyRectData = Array [1..NumOfRects] of Rect; {Graphics structure;     }
      MyRectPtr = ^MyRectData;                    {  an array of rectangles}
      MyRectHdl = ^MyRectPtr;

VAR
      hTE:  TEHandle;                 {TextEdit handle}
      hScroll,                        {Horizontal scroll bar}
      vScroll:  ControlHandle;        {Vertical scroll bar}
      MyWindow: WindowPtr;            {Document window}
      hdlScrollMenu: MenuHandle;      {Handle to the menu items}
      MyRect: MyRectHdl;              {Handle to array of rectangles}
      originalPart: INTEGER;          {1st part of the scroll bar hit}
      PageCorner,                     {Location of the upper left hand page corner}
      EventPoint: Point;              {Where an event took place}
      MyViewRect: Rect;               {display rectangle containing scrollable data}
      doneFlag,                       {Set TRUE when the user selects 'Quit'}
      showText,                       {Set TRUE when text can be scrolled}
      showGraphics : BOOLEAN;         {Set TRUE when graphics can be scrolled}

{-------------------------------------------------------------------------------}
PROCEDURE SetUpData;

{This procedure initializes two data structures; a TextEdit record and an array of
 rectangles.  Initially, only text and the vertical scrollbar will be displayed.}

Var MyString : StringHandle; {Temporary container for a string in the resource fork}
    counter : INTEGER;       {Counters must be local to the procedure}
    destRect : rect;         {Rectangle containing the larger-than-the-screen page}
BEGIN
{The TextEdit record is initialized by reading in a string from the application's
resource fork and then inserting it a number of times into the TextEdit record.}
  MyString := GetString (256); {Get some text to play around with}

{Set the view as the portrect less the vertical scrollbar area.  The TextEdit
destRect will be set to the current window width plus an arbitrary value.}
  MyViewRect := MyWindow^.portrect;
  destRect := MyViewRect;
  destRect.right := destRect.right + 300;
  PageCorner.h := -destRect.left;
```

```
  PageCorner.v := -destRect.top;
  MyViewRect.right := MyViewRect.right - 16; {16 = width of scrollbar}
  hTE := TENew (destRect, MyViewRect);

  HLock (Pointer (MyString));  {Can't move if we are going to point to the text}
  For counter := 1 to NumOfStrings DO {Create a TextEdit record full of the string}
    TEInsert (Pointer(Ord4(MyString^)+1), {move past the string's length byte}
              Length(MyString^^), hTE);
  HUnLock (Pointer (MyString));{Free to move again}

{Now, create a structure of rectangles.}
  MyRect := Pointer( NewHandle (Sizeof (MyRectData)));  {240 bytes }
  For counter := 1 to NumOfRects DO
    SetRect (MyRect^^[counter], counter*23, counter*20, counter*23+50, counter*20+50);

  showtext := TRUE;
  showgraphics := FALSE;
  ShowWindow (MyWindow); {Display the window and the text it contains}

  VScroll := GetNewControl (256, MyWindow); {vertical scrollbar}
  HScroll := GetNewControl (257, MyWindow); {horizontal scrollbar, not shown}

  CheckItem (hdlScrollMenu, vertical, TRUE);
  CheckItem (hdlScrollMenu, textItem, TRUE)
END; {of SetUpData}


{-------------------------------------------------------------------------------}
PROCEDURE GrafUpdate(whatpart : rect);
{This is roughly the equivalent of what TEUpdate does with text.  The upper left hand
corner of the page is moved up and to the left to simulate a view further down and
to the right.  To more accurately resemble a Toolbox routine like TEUpdate, this
procedure should also preserve the current clip region and origin.}
var count : INTEGER;
    dummyrect : rect;
BEGIN
  SetOrigin (PageCorner.h, PageCorner.v); {negative moves the origin left, up}
  OffsetRect (whatpart, PageCorner.h, PageCorner.v); {also move the update rectangle}
  ClipRect (whatpart); {only redraw the portion that the user requests}
  FOR count := 1 to NumOfRects DO
{Redraw the object if it intersects the update rectangle}
    IF SectRect (MyRect^^[count], whatpart, dummyRect)
    THEN FrameRect(MyRect^^[count]);
  SetOrigin (0,0); {reset the origin back to the upper left hand corner}
  ClipRect (MyWindow^.PortRect); {reset the clip region to the entire window}
END; {of GrafUpdate}

{-------------------------------------------------------------------------------}
PROCEDURE ScrollBits;
{This routine scrolls horizontally and vertically both graphics and text.  If you are
 only scrolling text, only the TEScroll is required.  If you are only scrolling
 graphics, then only the ScrollRect and GrafUpDate is needed.}

VAR vChange, hChange, vScrollValue, hScrollValue: INTEGER;
    AnUpdateRgn: RgnHandle;

BEGIN
  vScrollValue := GetCtlValue (vScroll); {these values will be used a lot so they are
  hScrollValue := GetCtlValue (hScroll); {read into local (temporary) variables}

{find the vertical and horizontal change}
  vChange := PageCorner.v - vScrollValue; {the vertical difference}
  hChange := PageCorner.h - hScrollValue; {the horizontal difference}

{record the values for next time}
  PageCorner.v := vScrollValue;
  PageCorner.h := hScrollValue;
```

```
{for pure text, only a TEScroll is required}
  IF showText AND NOT showGraphics THEN TEScroll (hChange, vChange, hTE);

{For graphics, a ScrollRect will move the visible bits on the screen.  The
 region returned by ScrollRect indicates what part of the window needs to
 be updated.}
  IF showGraphics THEN
  BEGIN
    AnUpdateRgn := NewRgn;
    ScrollRect (MyViewRect, hChange, vChange, AnUpdateRgn);

{This draws the new text.  The clipping is necessary because normally
 TextEdit redraws the entire character height and perhaps only a partial
 character scroll was done.  Since TextEdit erases before it draws, the text,
 if any, is drawn before the graphics.}
    IF showText THEN WITH hTE^^.destrect DO
    BEGIN
      left := -hScrollValue;
      top := -vScrollValue;
      ClipRect (AnUpdateRgn^^.rgnbbox);
      TEUpdate (AnUpdateRgn^^.rgnbbox, hTE);
      ClipRect (MyWindow^.portrect)
    END; {of showText}

    GrafUpdate (AnUpdateRgn^^.rgnbbox);  {This fills in the newly exposed region}
    DisposeRgn (AnUpdateRgn)
  END {of showGraphics}
END; {of ScrollBits}

{-------------------------------------------------------------------------------}
PROCEDURE TrackScroll(theControl: ControlHandle; partCode: INTEGER);
{This routine adjusts the value of the scrollbar.  A reasonable change would
 be to adjust the minimum scroll amount to equal the text's lineheight.}
Var amount, StartValue : INTEGER;
    up : BOOLEAN;
BEGIN
  up := partcode IN [inUpButton, inPageUp]; {TRUE if scrolling page up}
  StartValue := GetCtlValue (theControl);  {the initial control value}

  IF {the scrollbar value is decreased, and it is not already at the minimum}
     ((up AND (StartValue > GetCtlMin (theControl)))
  OR {the scrollbar value is increased, and it is not already at the maximum}
     ((NOT up) AND (StartValue < GetCtlMax (theControl))))
  AND {to prevent tracking as the page up or down area disappears}
     (originalPart = partCode)
  THEN
  BEGIN
    IF up THEN amount := -1 ELSE amount := 1;  {set the direction}
    IF partCode IN [inPageUp, inPageDown] THEN
    BEGIN
      {change the movement to a full page}
      WITH MyViewRect DO
      IF theControl = VScroll
      THEN amount := amount * (bottom - top)
      ELSE amount := amount * (right - left)
    END; {of partCode}
    SetCtlValue(theControl, StartValue+amount);
    ScrollBits
  END
END; {of TrackScroll}

{-------------------------------------------------------------------------------}
PROCEDURE MyControls;  {respond to a mouse down event in one of the controls}
Var dummy: INTEGER;
    theControl: ControlHandle;
BEGIN
  originalPart := FindControl (EventPoint, MyWindow, theControl); {returns control and part}
  IF originalPart = inThumb THEN
```

```
  BEGIN
    {Thumb is tracked until it is released; then the bits are scrolled}
    dummy := TrackControl (theControl, EventPoint, NIL);
    ScrollBits
  END {of whichpart}
  {for the arrows and the page changes, scroll while the mouse is held down}
  ELSE dummy := TrackControl (theControl, EventPoint, @TrackScroll)
END; {of Mycontrols}

{-------------------------------------------------------------------------------}
PROCEDURE MainEventLoop; {respond to menu selections, the scrollbars, and update events}
VAR myEvent: EventRecord;                {All of the information about the event}
    menuStuff: RECORD CASE INTEGER OF
      1 : (menuResult : LONGINT);        {Information returned by MenuSelect}
      2 : (theMenu,                      {Which menu was selected}
           theItem : INTEGER)            {Which item within the menu}
    END; {of menuStuff}
    checked : BOOLEAN;                   {Is the menu item checked}
    MarkChar : Char;                     {The checkmark character}
    tempWindow: WindowPtr;
    tempRect : Rect;

BEGIN
  REPEAT
    checked := GetNextEvent(everyEvent,myEvent); {checked here is ignored}
    CASE myEvent.what OF
    mouseDown:
      BEGIN {the user pressed or is holding the mouse button down}
        CASE FindWindow(myEvent.where,tempWindow) OF

          inMenuBar: WITH menuStuff DO
          BEGIN {the mouseDown was in the menu bar}
            menuResult := MenuSelect (myEvent.where);
            CASE theMenu OF
              FileMenu: doneFlag := TRUE;     { Quit }
              ScrollMenu:
              BEGIN
{The items in the menu are used to keep track of the user has chosen thus far. These
 lines toggle the checkmark in the menu and leave the result in the variable checked.}
                GetItemMark (hdlScrollMenu, theItem, markChar);
                checked := markChar <> Chr(checkmark);
                CheckItem (hdlScrollMenu, theItem, checked);

{Any selection will cause some part of the screen to be redrawn.  The selection that
the user makes causes some part of the screen to become invalid.}
                IF (theItem = textItem) OR (theItem = graphicsItem)
                THEN InvalRect(MyViewRect);

                CASE theItem OF

                  horizontal:
                  BEGIN
                    InvalRect (HScroll^^.contrlrect);
                    IF checked THEN
                    BEGIN
                      ShowControl(HScroll);
                      MyViewRect.bottom := HScroll^^.contrlrect.top
                    END {checked}
                    ELSE
                    BEGIN {not checked}
                      HideControl(HScroll);
                      MyViewRect.bottom := HScroll^^.contrlrect.bottom
                    END {not checked}
                  END; {horizontal}

                  vertical:
                  BEGIN
                    InvalRect (VScroll^^.contrlrect);
```

```
                    IF checked THEN
                    BEGIN
                      ShowControl(VScroll);
                      MyViewRect.right := VScroll^^.contrlrect.left
                    END {checked}
                    ELSE
                    BEGIN {not checked}
                      HideControl(VScroll);
                      MyViewRect.right := VScroll^^.contrlrect.right
                    END {not checked}
                  END; {vertical}

                  textItem: WITH hTE^^.destrect DO
{since we have dereferenced the destrect, no calls in the scope of this WITH should
 cause a memory compaction}
                  BEGIN
                    showText := checked;
                    IF checked then
                    BEGIN
                      top := -GetCtlValue(vScroll);
                      left := -GetCtlValue(hScroll);
                      hTE^^.viewrect := MyViewRect
                    END {of checked}
                  END; {of textItem}

                  GraphicsItem: showGraphics := checked;

                END {of CASE}
              END {of inMenuBar}
            END; {of FindWindow CASE}
            HiliteMenu(0)
          END; {of mouseDown}

        inContent:
{The rectangles making up the controls are the part of the window outside the 'view'}
          BEGIN
            EventPoint := MyEvent.where;
            GlobalToLocal (EventPoint);
            IF NOT PtInRect (EventPoint, MyViewrect) THEN MyControls
          END {in Content}
        END {of CASE}
      END; {of mouseDown}

    updateEvent:
{In response to InvalRects, the appropriate text or graphics is erased and redrawn.
 The BeginUpdate causes the VisRgn to be replaced by the intersection of the VisRgn
 and the UpdateRgn.}
      BEGIN
        BeginUpdate (MyWindow);
        EraseRect (MyWindow^.portrect); {start with a clean slate}
        IF showText THEN TEUpdate (MyWindow^.VisRgn^^.Rgnbbox, hTE);
{Call GrafUpdate with the intersection, if any, of the VisRgn and the view}
        IF showGraphics AND SectRect (MyWindow^.VisRgn^^.Rgnbbox, MyViewRect,
          tempRect) THEN GrafUpdate (tempRect);
        EndUpdate (MyWindow)
      END {of updateEvent}

    END {of event CASE}
  UNTIL doneflag
END;

{------------------------------------------------------------------------------}
BEGIN
  InitGraf (@ThePort); {initialize QuickDraw}
  InitWindows;         {initialize Window Manager; clear desktop and menubar}
  InitFonts;           {initialize Font Manager}
  FlushEvents (everyEvent, 0); {throw away any stray events}
  TEInit;              {initialize TextEdit}
```

```
  InitMenus;            {initialize Menu Manager}
  hdlScrollMenu := GetMenu(FileMenu); {(hdlScrollMenu is ignored)}
  InsertMenu (hdlScrollMenu,0);
  hdlScrollMenu := GetMenu(ScrollMenu);
  InsertMenu (hdlScrollMenu,0);
  DrawMenuBar;
  DoneFlag := FALSE;    {user 'Quit' flag}
  MyWindow := GetNewWindow (256, NIL, Pointer (-1)); {get window to work within}
  SetPort (MyWindow);   {point to window}
  TextFont (applFont); {select default application font}
  SetUpData;            {initialize user data and controls}
  InitCursor;           {change the watch into an arrow}
  MainEventLoop         {handle events until we are through}
END.
```

```
{ File -- Example code for printing, reading and writing files, and Text Edit }
{      -- by Cary Clark, Macintosh Technical Support }

PROGRAM MyFile;
{ Please read 'more about File,' included on the Mac Master disk. }

{SDECL BUG}
{SSETC BUG := 1}
{One good way of debugging code is to write status information to one of the
 serial ports.  Even while debugging code which uses one of the ports, the other
 can be used for transmitting information to an external terminal.

 In this program, the compile time variable BUG is set to either -1, 0 or 1
 according to the extent of the debugging information required.  Since compile
 time variables or constants are used, setting a single flag should cause the
 resulting program to have no more code than is required by the debugging level
 requested.

 If BUG is set equal to -1, then no debugging information appears;  this is as you
 would want the end user to see your product.

 BUG set to 0 provides an additional menu bar called 'debug' that can display the
 amount of memory available, compact memory, and discard segments and resources
 resident in memory.  You can do something similar to display some debugging
 information on the Mac itself if you do not have a terminal, but the penalty here
 is that you may spend much of your time debugging the code which is intended to
 debug some other part of the program.  Obviously, creating and maintaining a
 window on a screen full of other windows in untested code is a difficult thing to
 do.

 BUG set to 1 adds an additional item to the 'debug' menu that writes various runtime
 information to an external terminal.  This is the preferred method of debugging,
 since it does not interfere with the Macintosh display.  Even if you do not have
 a separate terminal, you can use the LISA terminal program to act as one.  Since
 writing a lot of debugging information to a serial port can slow the program down,
 I would recommend a way of turning the information on and off.  In this program,
 the variable DEBUG is set to true or false in the beginning of one of the first
 procedures executed, SETUP, to provide debugging information.  The DEBUG variable
 may also be set by the bottom item on the rightmost menu.}

{SU-} {Turn off the Lisa Libraries.  This is required by Workshop.}
{SX-} {Turn off stack expansion.  This is a Lisa concept, not needed on Mac.}

{SIFC BUG > -1}
  {SD+} {Put the procedures name just after it in the code, to help in debugging}
  {SR+} {Turn on range checking.  Violating the range at runtime will produce a
         check exception.}
{SELSEC}
  {SD-} {Do not include the procedure name in the 'production' code}
  {SR-} {Turn off range checking.}
{SENDC}

USES  {SU Obj/QuickDraw   } QuickDraw,
      {SU Obj/OSIntf      } OSIntf,
      {SU Obj/ToolIntf    } ToolIntf,
      {SU Obj/PackIntf    } PackIntf,
      {SU Obj/MacPrint    } MacPrint;

CONST
      appleMenu = 1;
      FileMenu = 2;
      EditMenu = 3;
      DebugMenu = 4;

{These constants are declared for this application to distinguish between the
```

```
various types of windows that it can create.  The number is stored in the window
field windowkind.}
     MyDocument = 8;
     Clipboard = 9;
     FreeMemory = 10;

{See the file Misc:Fileasm about the constants below.
 In this example program, I only use the first two.}
     TEScrpLength = 0;     {the length of the private TextEdit scrap}
     TEScrpHandle = 1;     {the handle to the private TextEdit }
     dlgFont      = 2;     {the font used inside alerts and dialogs}
     ScrVRes      = 3;     {screen vertical resolution (dots/inch)}
     ScrHRes      = 4;     {screen horizontal resolution (dots/inch)}
     doubleTime   = 5;     {double click time in 4/60's of a second}
     caretTime    = 6;     {caret blink time in 4/60's of a second}
     ANumber      = 7;     {the active alert}
     ACount       = 8;     {the alert stage level}

{SIFC BUG = -1}
     lastMenu = 3;       { number of menus w/o debug}
{SELSEC}
     lastMenu = 4;       { number of menus w/ debug}
{SENDC}

{SIFC BUG < 1}
     debug = FALSE;      { compiler will discard code after 'If debug ...'}
{SENDC}

TYPE ProcOrFunc = (proc, func, neither);
     edset = set of 1..9;
     appParms = RECORD                        { params set up by Finder at launch }
                  message: INTEGER;
                  count:   INTEGER;           { how many icons did the user select }
                  vRefNum: INTEGER;           { for each, the volume reference #, }
                  fTYPE:   resType;           {  the file type, }
                  vByte:   INTEGER;           {  the version number (should be 0) }
                  fName:   Str255;            {  and the name.  See SetUp for use. }
                END;
     pAppParms = ^appParms;

     MyData = RECORD {each document window keeps a handle to this in WRefCon}
                TERecord: TEHandle;     {the text associated with this document}
                FileVolume: Integer;    {which volume, if loaded from disk}
                changed: Boolean;       {the document is 'dirty'}
                titled:  Boolean;       {the document has never been saved to disk}
              END;
     MyDataPointer = ^MyData;
     MyDataHandle = ^MyDataPointer;
```

```
  {<<< this little beauty does a form feed when you print this out.
       Copy and Paste it to move it to your source code}
{Here are a ton of global variables.  This is not a good programming example.
 You professionals, of course, will keep the number of globals in your own
 programs to a much smaller number than shown here.}

{these first six values are changed as windows are activated}
VAR MyWindow: WindowPtr;
    MyPeek: WindowPeek;              {MyPeek is the same as MyWindow}
    WindowData: MyDataHandle;        {this record is pointed to by the WRefCon.}
    hTE:  TEHandle;                  {The active text edit handle}
    vScroll:  ControlHandle;         {The active vertical scroll bar.}
    topline:  integer;               {the value of VScroll, also the visible top line.}

    printhdl: THPrint;               {initialized in SetUp, used by MyPrint}
    myMenus: ARRAY
      [1..lastMenu] OF MenuHandle;   {Handles to all of the menus}
    growRect,                        {contains how big and small the window can grow}
    dragRect:  Rect;                 {contains where the window can be dragged}
    tempwindow: WindowPtr;           {window referenced by GetNextEvent (bad pgmming.)}
    theChar:  CHAR;                  {keyboard input goes here}
    myPoint: Point;                  {the point where an event took place}
    laststate: integer;              {last scrap state, to see if it has changed}
    doneFlag:  BOOLEAN;              {set when the user quits the program}
    myEvent:  EventRecord;           {returned by GetNextEvent}
    scrapwind:  WindowPtr;           {the ClipBoard window, which contains the scrap}
    iBeamHdl:  CursHandle;           {the text editing cursor}
    watchHdl:  CursHandle;           {the wait cursor}
    windownum: LongInt;              {the # of untitled windows opened}
    windowpos: LongInt;              {the # of windows opened}
    MyFileTypes: SFTypeList;         {same as txtfile, in a format for Standard File}
    firstchar: Integer;              {position of first character on top visible line}
    printflag: boolean;              {the user selected 'Print ' from the File menu}
    finderprint: boolean;            {the user selected 'Print' from the Finder}
    Dlogptr: DialogPtr;              {the dialog box used when printing from Finder}
    printing: boolean;               {printing is currently in progress}
    printport: grafptr;              {port preserved during background printing}
    numfiles: integer;               {number of files selected in finder}
{SIFC BUG > -1}
    FreeWind:  WindowPtr;            {the free memory window}
    oldmem: LongInt;                 {the last amount of free memory}
{SENDC}

{SIFC BUG = 1}
    debug: boolean;
{SENDC}
    debugger: text;                  {the external terminal file}
    extdebughdl: StringHandle;       {the menu entry}
    lf: char;                        {chr(10), linefeed}
```

```
{-------------------------------------------------------------------------------}
FUNCTION GlobalAddr (routineAddr: INTEGER) : Ptr;                    EXTERNAL;
FUNCTION GlobalValue (valueAddr: INTEGER) : LongInt;                 EXTERNAL;
{these routines, for now, allows us to retrieve where the TextEdit private scrap
 is, and allow us to set its size.  They are defined in Misc:FileAsm.}

PROCEDURE AutoScroll;                                                EXTERNAL;
{this assembly routine is called by the innerds of TextEdit when the user drags
 a selection range outside of the current window.  Needs work.}

PROCEDURE MainEventLoop; Forward;  {this is declared forward so the printing can
 take the main event loop as a procedure to execute while it is idleing}

FUNCTION MyGrowZone (cbNeeded: Size) : Size; Forward; {this is declared forward so
 that it can be resident in the blank segment, which is always loaded, and still
 be referenced by the SetUp procedure}

{SS Utilities}
{-------------------------------------------------------------------------------}
PROCEDURE DebugInProc (prockind: ProcOrFunc; where: str255; location: ptr);
{This procedure writes the executing routine's name and location in memory on the
 external terminal.  The location is especially important in a program like this
 that has segments.}
BEGIN
{SIFC BUG = 1}
  Write (debugger, 'in ');
  IF prockind = proc THEN Write (debugger, 'Procedure ');
  IF prockind = func THEN Write (debugger, 'Function ');
  Writeln (debugger, where, ' @ ', ord4(location), lf)
{SENDC}
END;
```

```
{-------------------------------------------------------------------------------}
PROCEDURE InSystemWindow;
VAR DScrap: PScrapstuff;
    tempport: grafptr;
BEGIN
{for desk accessories, service them with a SystemClick.  Also, check to see if they
 have changed the scrap.  If so, create an update event to redraw the clipboard.}
  if debug then debuginproc (proc, 'InSystemWindow', @InSystemWindow);
  SystemClick(myEvent, tempWindow);
  DScrap := InfoScrap;
  If (DScrap^.scrapState <> LastState) and (ScrapWind<>NIL) then
  BEGIN
    Getport (tempport);
    Setport (scrapwind);
    InvalRect (scrapwind^.portrect);
    Setport (tempport)
  END
END;

{-------------------------------------------------------------------------------}
PROCEDURE SetScrollMax;
Type txt= packed array [0..32000] of 0..255;
Var cr : integer;
    txtptr : ^txt;
    max: integer;
BEGIN
{This adjusts the scroll value so that the scroll bar range is not allowed to exceed
 the end of the text.  Also, the scroll bar is disabled if the max is set equal to
 the min, which is zero.  The formula for determining the range is somewhat complex.
 Sorry.}
  if debug then debuginproc (proc, 'SetScrollMax', @SetScrollMax);
  With hTE^^, hTE^^.viewrect DO
  BEGIN
    txtptr := pointer (htext^);
    cr := 0;
    {SR-}      {we want to be able to exceed the 32000 limit}
    if teLength > 0 then if txtptr^[teLength-1] = 13 then cr := 1;
    {SIFC BUG > -1}
    {SR+}
    {SENDC}
    max := nLines + cr - (bottom - top+1) DIV lineHeight;
    if max < 0 then max := 0;
    SetCtlMax (VScroll, max);
    if debug then Writeln (debugger, 'vscrollmax =', max,lf);
    topline := -destrect.top DIV lineheight;
    SetCtlValue (vscroll, topline);
    if debug then Writeln (debugger, 'topline =',topline,lf)
  END;
END;

{-------------------------------------------------------------------------------}
PROCEDURE ScrollText (showcaret: boolean);
{called to either show the caret after an action like 'Copy';
 also called to adjust the text within the window after the window is resized.  The
 same formula used in SetScrollMax is used here as well.  Don't worry about how this
 works, too much.  This possibly could be made much simpler.}
Type txt= packed array [0..32000] of 0..255; {we'll defeat this limit later}
Var bottomline, viewlines, SelLine, scrlAmount, numlines, blanklines, newtop
     : integer;
    txtptr: ^txt;
BEGIN
  if debug then DebugInProc (proc, 'ScrollText', @ScrollText);
  With HTE^^ DO
  BEGIN
    scrlAmount := 0;
    txtptr := Pointer(hText^);
    numlines := nlines;  {if the last character is a carriage return, add 1 to numlines}
```

```
{SR-}      {we want to be able to exceed the 32000 limit}
if teLength>0 then if txtptr^[teLength-1] = 13 then numlines := numlines + 1;
{SIFC BUG > -1}
{SR+}
{SENDC}
With HTE^^.viewrect DO
 viewlines := (bottom - top+1) DIV lineHeight;  {don't count partial lines}
topline := -destrect.top DIV lineheight;
bottomline := topline + viewlines - 1;
if debug then
BEGIN
  Write (debugger, 'nlines=',nlines:4,'; topline=',topline:4);
  Writeln (debugger, '; numlines=',numlines:4,'; bottom=',bottomline:4,lf);
  Writeln (debugger, 'viewlines=',viewlines:4,'; showcaret=',showcaret,lf)
END;
IF showcaret
THEN
BEGIN
  selLine := 0;
  While (selLine+1 < nlines) AND (selstart >=linestarts[selLine+1]) DO
   selLine := selLine + 1;
 {if selstart = selend is @ a cr, then add 1 to selstline}
  If (selstart = selend) AND (selstart > 0) then
{SR-}
  if (txtptr^[selstart-1] = 13) then selLine := selLine + 1;
{SIFC BUG > -1}
{SR+}
{SENDC}
  if debug then
  BEGIN
    Write (debugger, 'selstart=',selstart:5,'; selLine=',selLine:5);
    if selstart > 0 then
{SR-}
     Writeln (debugger, '; txtptr^[selstart-1] = 13 is ',txtptr^[selstart-1] = 13,lf)
{SIFC BUG > -1}
{SR+}
{SENDC}
  END;
  If SelLine > bottomline THEN
  BEGIN
    scrlAmount :=  bottomline - SelLine;
    If numlines - SelLine > viewlines DIV 2
    THEN scrlAmount := scrlAmount - viewlines DIV 2
    ELSE ScrlAMount := ScrlAmount - numlines + SelLine + 1
  END;
  If SelLine < topline THEN
  BEGIN
    scrlAmount := topline - SelLine;
    If selLine  > viewlines DIV 2
    THEN scrlAmount := scrlAmount + viewlines DIV 2
    ELSE ScrlAMount := ScrlAmount + selLine
  END
END;
if scrlAmount = 0 then
BEGIN
  blanklines := viewlines - numlines + topline;
  if blanklines < 0 then blanklines := 0;
  If (blanklines > 0) AND (topline > 0) THEN
  BEGIN
    scrlAmount := blanklines;
    If scrlAmount > topline THEN scrlAmount := topline
  END;
  if NOT showcaret then
  BEGIN
    newtop := 0;
    While (newtop+1 < nlines) AND (firstchar >= linestarts[newtop+1]) DO
     newtop := newtop + 1;
    if (newtop <> topline) AND (ABS(newtop - topline) > ABS(scrlAmount)) then
```

```
        scrlAmount := topline - newtop
      END
    END;
    if debug then
    BEGIN
      Write (debugger, 'newtop=',newtop:4,'; blanklines=',blanklines:4);
      Writeln (debugger, '; newtop - topline=',newtop - topline,lf)
    END;
    If scrlamount <> 0 THEN
    BEGIN
      If selstart = selend then TEDeactivate (hTE);
      TEScroll (0, scrlAmount * lineheight, hTE);
      If selstart = selend then TEActivate (hTE)
    END;
    if debug then Writeln (debugger, 'scrlAmount=',scrlAmount:4,lf);
    SetScrollMax
  END
END;
```

```
{-------------------------------------------------------------------------------}
PROCEDURE ToggleScrap;
Var temppeek: windowpeek;
    getwhich: integer;
    showhidehdl: StringHandle;
BEGIN
{The clipboard comes and goes, here.  The last item in the editmenu is alternately
 made to read, 'Show Clipboard' and 'Hide Clipboard'.}
  if debug then DebugInProc (proc, 'ToggleScrap', @ToggleScrap);
  If ScrapWind = NIL then {make it appear}
  BEGIN
    scrapwind := GetNewWindow (257, NIL, Pointer (-1));
    Temppeek := pointer (scrapwind);
    Temppeek^.windowkind := Clipboard;
    SetPort (scrapwind);
    InvalRect (scrapwind^.Portrect);
    GetWhich := 263 {hide clipboard}
  END
  else    {make it disappear}
  BEGIN
    DisposeWindow (scrapwind);
    Scrapwind := NIL;
    GetWhich := 262 {show clipboard}
  END;
  showhidehdl := GetString (getwhich);
  Hlock (Pointer(showhidehdl));
  SetItem (myMenus[EditMenu], 9, showhidehdl^^);
  Hunlock (Pointer(showhidehdl))
END;

{SIFC BUG > -1}
{-------------------------------------------------------------------------------}
PROCEDURE ToggleFree;
Var temppeek: windowpeek;
    getwhich: integer;
    showhidehdl: StringHandle;
BEGIN
{just about the same as ToggleClipboard, above.  This is just for debugging fun.}
  if debug then DebugInProc (proc, 'ToggleFree', @ToggleFree);
  If FreeWind = NIL then {make it appear}
  BEGIN
    Freewind := GetNewWindow (258, NIL, Pointer (-1));
    Temppeek := pointer (Freewind);
    Temppeek^.windowkind := FreeMemory;
    SetPort (Freewind);
    InvalRect (Freewind^.Portrect);
    GetWhich := 265;
  END
  else    {make it disappear}
  BEGIN
    DisposeWindow (Freewind);
    Freewind := NIL;
    GetWhich := 264
  END;
  showhidehdl := GetString (getwhich);
  Hlock (Pointer(showhidehdl));
  SetItem (myMenus[DebugMenu], 1, showhidehdl^^);
  HUnlock (Pointer(showhidehdl))
END;
{SENDC}

{-------------------------------------------------------------------------------}
PROCEDURE SetViewRect;
BEGIN
{text edit's view rect is inset in the content of the window, to prevent it from
 running into the lefthand side or the scroll bar.}
  if debug then DebugInProc (proc, 'SetViewRect', @SetViewRect);
```

```
  With hTE^^.viewrect DO
  BEGIN
    hTE^^.viewrect := MyWindow^.portRect;
    left  :=  left +4;
    right := right -15
  END
END;
```

```
{-------------------------------------------------------------------------------}
PROCEDURE MoveScrollBar;
BEGIN
{When the window is resized, the scroll bar needs to be stretched to fit.}
  if debug then DebugInProc (proc, 'MoveScrollBar', @MoveScrollBar);
  WITH MyWindow^.portRect DO
  BEGIN
    HideControl(vScroll);
    MoveControl(vScroll,right-15,top-1);
    SizeControl(vScroll,16,bottom-top-13);
    ShowControl(vScroll)
  END
END;

{-------------------------------------------------------------------------------}
PROCEDURE GrowWnd;
{ Handles growing and sizing the window and manipulating the update region. }
VAR longResult:  LongInt;
    height, width, newvert, oldstart:  INTEGER;
    tRect, oldportrect: Rect;
BEGIN
  if debug then DebugInProc (proc, 'GrowWnd', @GrowWnd);
  longResult := GrowWindow(MyWindow,myEvent.where,growRect);
  IF longResult = 0 THEN EXIT(GrowWnd);
  Setcursor (watchhdl^^); {because the word wrap could take a second or two}
  height := HiWord(longResult); width := LoWord(longResult);
  SizeWindow(MyWindow,width,height,TRUE);  { Now draw the newly sized window. }
  InvalRect (MyWindow^.portrect);
  If MyPeek^.windowkind = MyDocument then {it's not the clipboard}
  BEGIN
    MoveScrollBar;
    With MyWindow^.portRect DO
    BEGIN
      width := right-left-19;
      height := bottom-top
    END;
    With HTE^^ DO
    BEGIN
      destrect.right := destrect.left + width;
      viewrect.right := viewrect.left + width;
      viewrect.bottom := viewrect.top + height;
      firstchar := hTE^^.linestarts [topline];
      TECalText (hTE); {re-wrap the text to fit the new screen.}
      {if the rectangle is grown such that there is now blank space on the bottom
      of the screen, backpedal the screen to fill it back up, if there is enough
      scrolled off the screen to do so.  Otherwise, the first character in the top line on
      the screen should continue to be somewhere on the top line after resizing}
      ScrollText (FALSE);
    END
  END
END;     { of GrowWnd }

{-------------------------------------------------------------------------------}
PROCEDURE MyActivate;
VAR tRect : rect;
BEGIN
{activate events occur when one window appears in front of another.  This takes care
 of hiliting the scroll bar and deactivating the insertion caret or the text
 selection.}
  if debug then DebugInProc (proc, 'MyActivate', @MyActivate);
  MyWindow := Pointer (MyEvent.message);
  MyPeek := Pointer (MyWindow);
  If MyPeek^.windowkind in [MyDocument,Clipboard] then
  BEGIN {redraw the scrollbar area, if a document or the clipboard}
      SetPort (MyWindow);
      tRect := MyWindow^.portRect; tRect.left := tRect.right-16;
      InvalRect(tRect)
```

```
  END;
  If MyPeek^.windowkind = MyDocument then
  BEGIN {make global variables point to the information associated with this window}
    WindowData := Pointer (GetWRefCon (MyWindow));
    VScroll := Pointer (MyPeek^.ControlList);
    hTE := WindowData^^.TERecord;
    IF ODD (myEvent.modifiers)
    THEN
    BEGIN {this window is now top most}
      TEActivate(hTE);
      ShowControl (VScroll);
      topline := GetCtlValue (VScroll)
    END
    ELSE
    BEGIN {this window is no longer top most}
      HideControl (VScroll);
      TEDeactivate(hTE);
      hTE := NIL {a document is no longer on top}
    END
  END
END;    { of activateEvt }
```

```
{-------------------------------------------------------------------------------}
PROCEDURE DialogueDeactivate;
var temprect: rect;
BEGIN
{This routine takes care of cases where, for instance, a modal dialog is about to
 pop up in front of all the other windows.  Since the Dialog Manager handles all
 activate events for you, you do not get a chance to 'turn off' the controls associated
 with the window.  This routine is called just before the dialog box makes its
 appearance, and takes care of the hiliting as if an activate event had occured.}
  if debug then DebugInProc (proc, 'DialogueDeactivate', @DialogueDeactivate);
  If hTE <> NIL then {for documents, only}
  BEGIN
    TEDeactivate(hTE);
    HideControl (VScroll);
    SetCursor (arrow)
  END;
  If (frontwindow <> NIL) AND (Mypeek^.windowkind in [MyDocument, Clipboard]) then
  BEGIN {this is a little kludgy, but it works.}
    Mypeek^.hilited := false; {DrawGrowIcon will now unhilite.}
    temprect := MyWindow^.PortRect;
    temprect.left := temprect.right - 15;
    Cliprect (temprect); {clipaway the horizontal scrollbar part}
    DrawGrowIcon (MyWindow);
    Cliprect (MyWindow^.PortRect);
    Mypeek^.hilited := true {fix things back}
  END
END;

{
```

```
}
{SS READFILE}
{-------------------------------------------------------------------------------}
Function ReadFile (VrefNo: integer; FName : str255) : boolean;
Var refNo, io : integer;
    logEOF: LongInt;
    errin: str255;

{-------------------------------------------------------------------------------}
  Procedure DiskRErr (io : integer);
  Var str: str255;
      readfromhdl, loadedhdl, strhdl: StringHandle;
      dummy: integer;
  BEGIN
{A generic error is reported to the user if something goes wrong.  Amazingly little can
 go wrong, since the user does not get the chance to do things like type file names,
 remove the disk himself, and so on.  About the only error that could happen is:

 an error occured while reading the disk (damaged media or hardware)

 Can you think of anything else?  A similar routine further down handles
 writing to disk.  Note that in both reading and writing, the entire file is handled
 by a single read/write call, and no 'disk buffer' needs to be specified by the
 programmer.}

    if debug then
    BEGIN
      DebugInProc (func, 'DiskRErr', @DiskRErr);
      Writeln (debugger, errin, ' err = ', io, lf)
    END;
    readfromHdl := GetString (267); {this says 'reading from'}
    loadedhdl := GetString (269); {this says 'loaded'}
    HLock (Pointer(readfromHdl));
    HLock (Pointer(loadedhdl));
    If io = IOErr then
    BEGIN
      strhdl := GetString (279); {this says 'IO error'}
      str := strhdl^^
    END else
    BEGIN
      NumToString (io, str);
      strhdl := GetString (280); {this is the generic 'ID ='}
      str := Concat (strhdl^^, ' ', str)
    END;
    Paramtext (readfromhdl^^, FName, loadedhdl^^, str);
    SetCursor (arrow);
    dummy := StopAlert (256, NIL); {discribe error to user in generic way.}
    HUnlock (Pointer(readfromHdl));
    HUnlock (Pointer(loadedhdl));
    Exit (readfile)
  END;

BEGIN
  if debug then
  BEGIN
    DebugInProc (func, 'ReadFile', @ReadFile);
    writeln (debugger, 'volume = ', vrefno, '; file = ', fname, lf)
  END;
  SetCursor (watchHdl^^);
  ReadFile := false;
  io := FSOpen (Fname, VRefNo, RefNo);
{SIFC BUG = 1} {these debugging statements are for the external terminal, only}
  errin := 'FSOpen';
{SENDC}
  If io <> 0 then DiskRErr (io);
  io := GetEOF (RefNo, logEOF);
{SIFC BUG = 1}
  errin := 'GetEOF';
```

```
{SENDC}
  If io <> 0 then DiskRErr (io);
  {add code here:  if file is too large, then notify user and truncate}
  SetHandleSize (hTE^^.hText, logEOF);
  if debug then if memerror<>0 then Writeln (debugger, 'memerr = ',memerror:4);
  io := FSRead (refNo, logEOF, hTE^^.hText^);
{SIFC BUG = 1}
  errin := 'FSRead';
{SENDC}
  If io <> 0 then DiskRErr (io);
  io := FSClose (refNo);
{SIFC BUG = 1}
    errin := 'FSClose';
{SENDC}
  If io <> 0 then DiskRErr (io);
  hTE^^.teLength := logEOF;
  if not finderprint then {if printing from the finder, no window or editing
   information is needed}
  BEGIN
    TESetSelect (0,0,hTE);
    TECalText (hTE);
    Invalrect (hTE^^.viewrect);
    SetScrollMax;
    WindowData^^.titled := true;
    WindowData^^.changed := false;
    WindowData^^.FileVolume := VRefNo
  END;
  ReadFile := True {everything worked out OK}
END;
```

```
{-------------------------------------------------------------------------------}
PROCEDURE MakeAWindow (str : str255; disk : boolean);
Var bounds: rect;
BEGIN
{A window is created here, and all associated data structures are linked to it}
  if debug then DebugInProc (proc, 'MakeAWindow', @MakeAWindow);
  windowpos := windowpos + 1; {this position it is created to on the screen}
  bounds.left :=  windowpos MOD 16 * 20 + 5;
  bounds.top := windowpos MOD 11 * 20 + 45;
  bounds.right := bounds.left + 200;
  bounds.bottom := bounds.top + 100;
  MyWindow := NewWindow(Nil, bounds, str, TRUE, 0, POINTER(-1), TRUE, 0);
  SetPort (MyWindow);
  Mypeek := Pointer (MyWindow);
  TextFont (NewYork);
  Mypeek^.windowkind := MyDocument; {a number > 8 identifies the type of window}
  hTE := TENew(MyWindow^.portRect, MyWindow^.portRect);
  WindowData := Pointer (NewHandle (8)); {1 handle, an integer, and 2 booleans}
  SetWRefCon (MyWindow, Ord4 (WindowData));
  WindowData^^.TERecord := hTE;
  SetViewRect;
  hTE^^.destrect := hTE^^.viewrect;
  WindowData^^.changed := false;
  WindowData^^.titled := false;
  vScroll := GetNewControl(256,MyWindow);
  MoveScrollBar;
  topline := 0;
  hTE^^.clikLoop := ord4(@AutoScroll)
END;
```

```
{-------------------------------------------------------------------------------}
PROCEDURE MyGetFile;
Var reply: SFReply;
    wher: point;
    temprect: rect;
    tempport:grafptr;
    copyIt, foundIt : boolean;    {if the name is already in use, this will be true}
    temppeek: Windowpeek;
    tempstr, oldfname: str255;
    strhdl: stringhandle;
    tempdata: MyDataHandle;
    temphdl: Handle;
BEGIN
{This calls Standard File to allow the user to choose the document on disk that she
 wishes to edit.}
  if debug then DebugInProc (proc, 'MyGetFile', @MyGetFile);
  wher.h := 90;
  wher.v := 100;
  {** experiment:  the system appears to want CDEF 1 loaded, so let's load it and
   make it nonpurgable before continuing}
  temphdl := GetResource ('CDEF',1);
  HNoPurge (temphdl);
  DialogueDeactivate;
  SFGetFile (wher, '', NIL, 1, MyFileTypes, NIL, reply);
  With Reply DO
  If good then
  BEGIN
    {check to see if this name already resides on a document window.  If so, change
     the title to 'Copy of   ' and remember to check it as untitled? after the readfile}
    foundit := false;
    oldfname := fname;
    Repeat
      temppeek := pointer(Frontwindow);
      copyIt := false;
      if temppeek <> NIL then
      Repeat
        GetWTitle (pointer (temppeek), tempstr);
        if tempstr = fname then
        BEGIN
          tempdata := Pointer(temppeek^.refCon);
          if tempdata^^.FileVolume = vrefnum then
          BEGIN
            copyIt := true;
            foundIt := true
          END
        END;
        temppeek := temppeek^.nextwindow
      Until (temppeek = NIL) or copyIt;
      strhdl := GetString (274);
      if copyIt then fname := Concat (strhdl^^,' ',fname);
    Until not copyIt;
    if foundIt then
    BEGIN
      Paramtext (fname,'','','');
      copyIt := (NoteAlert (258, NIL) = OK)
    END;
    if not foundIt or copyIt then
    BEGIN
      MakeAWindow (fname, TRUE);
      If ReadFile (vrefnum, oldfname)
      then
      BEGIN
        if foundIt then windowdata^^.titled := false
      END
      else
      BEGIN
        TEDispose (hTE);
```

```
        hTE := NIL;
        DisposHandle (Pointer (WindowData));
        If debug then Writeln (debugger, 'dispose WindowData; memerr = ', MemError, lf);
        DisposeWindow (MyWindow)
      END
    END
  END;
  HPurge (temphdl); {** the other half of the experiment (above)}
END;

{-------------------------------------------------------------------------------}
PROCEDURE OpenAWindow;
VAR s: str255;
    untitled: StringHandle;
BEGIN
{this creates a new window that is untitled and empty.}
  if debug then DebugInProc (proc, 'OpenAWindow', @OpenAWindow);
  {see if enough mem exists to open a window}
  NumToString(windownum, s);
  windownum := windownum + 1;
  untitled := GetString (256);
  HLock (Pointer(untitled));
  MakeAWindow (Concat (untitled^^, s), FALSE);
  HUnlock (Pointer(untitled))
END;
```

```
{SS WRITFILE}
{-------------------------------------------------------------------------------}
FUNCTION WriteFile (vRefNo: integer; fName : str255) : Boolean;
var refNo, io : integer;
    txtlength : longint;
    errin : str255;

{-------------------------------------------------------------------------------}
  PROCEDURE DiskWErr (io : integer);
  Var str:str255;
      writetoHdl, savedHdl, strhdl: StringHandle;
      dummy, errstr: integer;
  BEGIN
  {This is just about the same as DiskRErr (read).  Since a few more errors can
   happen during a write, the structure is just a little different}
    if debug then
    BEGIN
      DebugInProc (proc, 'DiskWErr', @DiskWErr);
      Writeln (debugger, errin, ' err = ', io, lf)
    END;
    {read resource for writeto}
    writetoHdl := GetString (268);
    {read resource for saved}
    savedHdl := GetString (270);
    Hlock (Pointer(writetoHdl));
    Hlock (Pointer(savedhdl));
    errstr := 0;
    Case io of
      DirFulErr : errstr := 276;
      DskFulErr : errstr := 275;
      IOErr : errstr := 279;
      FLckdErr : errstr := 277;
      VLckdErr, WPrErr : errstr := 278;
      Otherwise
      BEGIN
        NumToString (io, str);
        strhdl := GetString (280);
        str := Concat (strhdl^^, ' ',str)
      END
    END;
    if errstr <> 0 then
    BEGIN
      strhdl := GetString (errstr);
      str := strhdl^^;
      Hlock (Pointer(strhdl))
    END;
    Paramtext (writetoHdl^^,FName,savedhdl^^,str);
    SetCursor (arrow);
    dummy := StopAlert (256, NIL);
    HUnlock (Pointer(writetoHdl));
    HUnlock (Pointer(savedhdl));
    HUnlock (Pointer(strhdl));
    Exit (writefile)
  END;

BEGIN
{this isn't very different from read file.  The only complication is finding out
 if the file exists.  If it doesn't, create it.  Also, assign the information that
 the finder needs to properly associate it with this application.}
  if debug then DebugInProc (proc, 'WriteFile', @WriteFile);
  SetCursor (watchHdl^^);
  WriteFile := False;
  io := FSOpen(FName, VRefNo, refNo);
{SIFC BUG = 1}
  errin := 'FSOpen'; {once again, these only benefit the external debugger.}
{SENDC}
  If debug then Writeln (debugger, 'file RefNum =', refNo, lf);
```

```
  If io = {file not found Err} -43 then
  BEGIN
    io := Create (FName,VRefNo,'CARY','TEXT');
{SIFC BUG = 1}
    errin := 'Create';
{SENDC}
    If io <> 0 then DiskWErr (io);
    io := FSOpen(FName, VRefNo, refNo);
{SIFC BUG = 1}
    errin := 'FSOpen';
{SENDC}
    if debug then Writeln (debugger, 'file RefNum = ', refNo, lf);
    If io <> 0 then DiskWErr (io)
  END {Create}
  ELSE If io <> 0 then DiskWErr (io);
  With hTE^^ DO
  BEGIN
    txtLength := Ord4(teLength);
    HLock (hText);
    io := FSWrite (refNo, txtLength, hText^);
    HUnlock (hText)
  END;
  if debug then Write (debugger, '.');
{SIFC BUG = 1}
  errin := 'FSWrite';
{SENDC}
  If io <> 0 then DiskWErr (io);
  io := SetEOF (refNo, txtlength);
  if debug then Write (debugger, '.');
{SIFC BUG = 1}
  errin := 'SetEOF';
{SENDC}
 If io <> 0 then DiskWErr (io);
  io := FSClose (refNo);
  if debug then Write (debugger, '.');
{SIFC BUG = 1}
  errin := 'FSClose';
{SENDC}
  If io <> 0 then DiskWErr (io);
  io := FlushVol (NIL, VrefNo); {this is important; without it, if the program died
  (not possible as a result of a programming mistake, of course), the directory
   information on the disk would not be accurate.}
  if debug then Write (debugger, '.');
{SIFC BUG = 1}
  errin := 'FlushVol';
{SENDC}
  If io <> 0 then DiskWErr (io);
  if not windowdata^^.titled then
  BEGIN
    SetWTitle(MyWindow, FName);
    WindowData^^.filevolume := VRefNo
  END;
  WindowData^^.titled := true;
  WindowData^^.changed := false;
  WriteFile := True {everything is OK.}
END;
```

```
{-------------------------------------------------------------------------------}
FUNCTION MyPutFile (Filename: str255): Boolean;
Var reply: SFReply;
    wher: point;
    NameHdl: StringHandle;
    temprect: rect;
    tempport:grafptr;
BEGIN
{The user can select the name of the file that they wish to save the document with.}
  if debug then DebugInProc (func, 'MyPutFile', @MyPutFile);
  MyPutFile := False;
  NameHdl := GetString (257);
  wher.h := 100;
  wher.v := 100;
  Hlock (Pointer(namehdl));
  DialogueDeactivate;
  SFPutFile (wher, NameHdl^^, Filename, NIL, reply);
  HUnlock (Pointer(namehdl));
  With Reply DO
  BEGIN
    if debug then Writeln (debugger, 'reply.good = ', good, lf);
    If good then MyPutFile := WriteFile (vrefnum, fname)
  END;
  if debug then Writeln (debugger, 'release reserror = ', reserror, lf)
END;

{-------------------------------------------------------------------------------}
PROCEDURE CloseAWindow;
VAR itemhit: integer;
    DBoxPtr: DialogPtr;
    str,str1: str255;
    Goodwrite: Boolean;
    temprect: rect;
    NameHdl: StringHandle;
    NamePtr: ^Str255;
    typ: integer;
    itemhdl: handle;
    box:rect;
BEGIN
{All sorts of windows can be closed through this single routine, which is accessed
 by the user through the go-away box on the window, or the Close item in the File
 menu, or by quitting the program.}
  if debug then DebugInProc (proc, 'CloseAWindow', @CloseAWindow);
  MyPeek := Pointer (FrontWindow);
  Case Mypeek^.windowkind of
  MyDocument :
  BEGIN
    GetWTitle (MyWindow, str);
    itemhit := 0;
    If WindowData^^.changed then {give the user the chance to save his data before
     you throw it away.}
    BEGIN
      DialogueDeactivate;
      if doneflag then
      BEGIN
        NameHdl := GetString (266);
        str1 := NameHdl^^;
        if debug then Writeln (debugger, 'err = ', Reserror, lf);
      END
      ELSE str1 := '';
      Paramtext (str,str1,'','');
      ItemHit := CautionAlert (259, NIL)
    END;
    if debug then Writeln (debugger, 'itemhit = ', itemhit, lf);
    Goodwrite := false;
    if not windowdata^^.titled then str := '';
    If itemhit = OK {save} then
```

```
      if WindowData^^.titled
      then GoodWrite := WriteFile (WindowData^^.FileVolume, str)
      else Goodwrite := MyPutFile (str);
    If GoodWrite or (itemhit IN [0, 3] {discard}) then
    BEGIN
      TEDispose (hTE);
      hTE := NIL;
      DisposHandle (Pointer (WindowData));
      DisposeWindow (MyWindow)
    END;
    If itemhit = Cancel then doneflag := false
  END;
  Clipboard : ToggleScrap;
{SIFC BUG > -1}
  FreeMemory: ToggleFree;
{SENDC}
    OTHERWISE CloseDeskAcc (MyPeek^.windowkind) {can't be anything else}
  END {Case}
END;
```

```
{SS AboutMyPgm}
{-------------------------------------------------------------------------------}
PROCEDURE AboutMyEditor;
var str1hdl, str2hdl : stringHandle;
    MyWindow:  WindowPtr;
    width, height, counter: integer;
    newcount: longint;
    quit: boolean;
    txtinfo: fontinfo;
    temprect, trect1: rect;
    tempbits: bitmap;
    sz: size;
BEGIN
{this bit of fluff shows a wrong method of telling the user something about
 my program, but it was fun to do.}
  if debug then DebugInProc (proc, 'AboutMyEditor', @AboutMyEditor);
  DialogueDeactivate;
  str1hdl := GetString (259);
  if debug then Writeln (debugger, 'err = ', Reserror, lf);
  str2hdl := GetString (260);
  if debug then Writeln (debugger, 'err = ', Reserror, lf);
  HLock (Pointer(str1hdl));
  HLock (Pointer(str2hdl));
  MyWindow := GetNewWindow (256, NIL, Pointer (-1));
  SetPort (MyWindow);
  counter := 1;
  width := MyWindow^.portrect.right - MyWindow^.portrect.left;
  height := MyWindow^.portrect.bottom - MyWindow^.portrect.top;
  TextFont (NewYork);
  TextMode (srcCopy);
  quit := false;
  Repeat
    SystemTask;
    newcount := tickcount + 6;
    TextSize (counter);
    GetFontInfo (txtinfo);
    With txtinfo Do
    Begin
      MoveTo ((width - StringWidth (str1hdl^^)) DIV 2, height DIV 2 - descent - leading);
      DrawString (str1hdl^^);
      MoveTo ((width - StringWidth (str2hdl^^)) DIV 2, height DIV 2 + ascent);
      DrawString (str2hdl^^)
    End;
    If EventAvail (mDownMask+keyDownMask, MyEvent) then quit := true;
    counter := counter + 1;
    While newcount > tickcount Do;
  Until quit or (counter = 12);
  newcount := tickcount + 300; {5 seconds}
  while not quit and (tickcount < newcount) Do
  Begin
    SystemTask;
    If EventAvail (mDownMask+keyDownMask, MyEvent) then quit := true;
  End;
  temprect := MyWindow^.portrect;
  With txtinfo Do
  Begin
    temprect.top := height DIV 2 - ascent - descent - leading;
    temprect.bottom := height DIV 2 + ascent + descent
  End;
  trect1 := temprect;
  OffsetRect (trect1, 0, -trect1.top);
  tempbits.rowbytes := (width + 7) DIV 8;
  tempbits.bounds := trect1;
  With Txtinfo Do sz := Ord4 (tempbits.rowbytes * (ascent*2+descent*2+leading));
  tempbits.baseaddr := pointer (NewPtr (sz));
  if debug then Writeln (debugger, 'err = ', Memerror, lf);
  CopyBits (MyWindow^.portbits, tempbits, temprect, trect1, srcCopy, NIL);
```

```
  insetrect (trect1, 8, 0);
  temprect.top := temprect.top - 2;
  temprect.bottom := temprect.bottom + 2;
  while not quit and (trect1.right > width DIV 2) Do
  Begin
    SystemTask; {the clock still ticks!}
    CopyBits (tempbits, MyWindow^.portbits, trect1, temprect, srcCopy, NIL);
    If temprect.top > Mywindow^.portrect.top then
    Begin
      insetrect (trect1, 8, 0);
      insetrect (temprect, 0, -2)
    End else insetrect (trect1, 8, 2);
    If EventAvail (mDownMask+keyDownMask, MyEvent) then quit := true
  End;
  HUnlock (Pointer(str1hdl));
  HUnlock (Pointer(str2hdl));
  DisposPtr(Pointer(tempbits.baseaddr));
  if debug then Writeln (debugger, 'err = ', Memerror, 1f);
  DisposeWindow (mywindow)
END;
```

```
{SS MyPrint }
{-------------------------------------------------------------------------------}
PROCEDURE CheckButton;
var bool : boolean;
    item : integer;
    PrinterErrPtr : ^Integer;
BEGIN
  bool := GetNextEvent (mDownMask+keyDownMask, MyEvent);
  item := 0;
  if (myEvent.what = keydown) and (myEvent.message MOD 256 = 13) then item := 1
  else
  If IsDialogEvent (myEvent) then bool := DialogSelect (myEvent, dlogptr, item);
  If item = 1 then
   BEGIN
     PrinterErrPtr := Pointer(pPrGlobals);
     PrinterErrPtr^ := IprAbort;
   END
END;

{-------------------------------------------------------------------------------}
PROCEDURE MyPrint(finderFile:integer; filename: str255);
Const bottommargin = 20;      {amount of space on the margins of the page in pixels}
      leftmargin = 30;
      rightmargin = 10;

Var MyPPort: TPPrPort;
    txt: handle;
    txtptr: ptr;
    pglen, start, finish, counter, count2, loop, io, numpages: integer;
    temprect, tmprect2, pagerect: rect;
    status: TPrStatus;
    userOK, canceldialog: boolean;
    s: string[1];
    str: str255;
    numToGo, numdone: str255;
    temphdl: stringhandle;
    MyLngth: array [1..99] of integer;
Begin
{For heavyweight programmers only. All modes of printing are handled by Macprint.  The
 only things you have to do are:
   image each page, using QuickDraw (or something that uses QuickDraw);
   Do it once for the number of copies the user specified in draft mode only.

   You do not have to worry with:
     copies in normal or high res.
     which pages the user chose to print.
     tall, wide, etc.

  Remember, these Page Setup dialog is printer specific.  It will not always be the
  same, so don't write any code around it.

  The reason this program is heavily segmented is that printing normal or high-res
  on line takes gobs of memory (in this example, up to 25K.)  You may minimize the
  by omitting 1 line below and creating a spooled file instead.

  The finderprint boolean determines whether printing is has been selected while the
  user is running the application, or whether it was selected from the finder.  In the
  application, printing is done in the background.  From the finder, a simple dialog
  is presented instead.  Because printing takes a large amount of memory, up to 25K,
  background printing is only possible if the memory required by the foreground
  process can be kept to a minimum.  Since this program does not yet have strong
  memory full checking, you should set the debugging compile time variable DEBUG
  to -1, and remove MacsBug from the Mac disk, to give the program a realistic amount
  of free memory.  MacsBug, when active, can use up to 16K.

  Printing is not re-entrant.  If your main program loop is the print idle proc,
  as below, disable the Page Setup item and change 'Print ' to 'Stop Printing'
```

```
in the File menu.}

if debug then DebugInProc (proc, 'MyPrint', @MyPrint);
printflag := false;
if debug then
 writeln (debugger, 'finderPrint =', finderprint, '; finderFile =', finderfile, lf);
userOK := true;
If finderfile = 1 then
BEGIN
  SetCursor (arrow);
  userOK :=  PrJobDialog (PrintHdl)
END;
If userOK then
BEGIN
  {try to see if enough memory exists to
       1) duplicate the text portion of the te record
       2) allow the printing pieces to be resident
       3) allow the largest possible segment to be loaded by the main event loop
       if so, allow the printing to go on in the background.
       Otherwise, put up the 'press a button to cancel' dialog}
  SetCursor (watchhdl^^);
  if not finderprint then numfiles := 1;
  canceldialog := finderprint;
  if not canceldialog then
  BEGIN
    txt := NewHandle (hte^^.telength+16000);
    {this calculation should be made considering:
       the current font size
       the printing mode (draft, normal, hires)
       the textstyle overhead, if any
       blank segment overhead
       largest segment + largest local data
       global data overhead ---  16000 is a crude, unprofessional approximation}
    if txt = NIL then canceldialog := true
    else
    BEGIN
      disposHandle (txt);
      txt := hte^^.hText;
      ResrvMem (hte^^.teLength);
      io := HandToHand (txt);
    END
  END;
  if canceldialog then
  BEGIN
    NumToString (finderFile, numToGo);
    NumToString (numfiles, numdone);
    Paramtext (filename,numToGo,numdone,'');
    dlogptr := GetNewDialog (257, NIL, Pointer(-1));
    DrawDialog (dlogptr);
    printHdl^^.prJob.pIdleProc := @CheckButton;
    txt := hte^^.hText
  END
  else
  BEGIN
    temphdl := GetString (273); {change to 'Stop Printing'}
    HLock (pointer(temphdl));
    SetItem (myMenus[fileMenu], 8, temphdl^^);
    HUnlock (pointer(temphdl));
    printing := true;
    printHdl^^.prJob.pIdleProc := @MainEventLoop;
    GetPort (printport); {get the port to be restored at the top of the
     main event loop}
  END;
  {for now, approximate a full page}
  MyPPort := PrOpenDoc (PrintHdl, NIL, NIL);
  With hTE^^, printhdl^^.prinfo Do
  BEGIN
    pagerect := rpage;
```

```
      pagerect.left := pagerect.left + leftmargin;
      pagerect.right := pagerect.right - rightmargin;
      pagerect.bottom := pagerect.bottom - bottommargin
      - (pagerect.bottom - bottommargin) MOD lineheight {get rid of partial line};
      temprect := destrect;
      destrect := pagerect;
      TECalText (hTE)
    END;  {TECalText could cause the memory manager to move the hTE and PrintHdl
     handles.  So, the 'With' statement is required below; the alternative would
     be to use 1 'With' and 'HLock' the handles.  Note that 'With' is much more
     than a lexical convenience.  It actually causes the compiler to optimize code
     about the fields of hTE^^ and printhdl^^.prinfo}
    With hTE^^, printhdl^^.prinfo Do
    BEGIN
      tmprect2 := viewrect;
      pglen := (rpage.bottom - rpage.top - bottommargin) DIV lineheight;
      finish := nlines;
      start := 0;
      counter := 1;
      While start < finish Do
      Begin
        IF finish - start > pglen
        THEN MyLngth[counter] := linestarts[start + pglen] - linestarts[start]
        ELSE MyLngth[counter] := teLength - linestarts[start];
if debug then
BEGIN
  Writeln (debugger, 'MyLngth[', counter:1, '] = ', MyLngth[counter]:5, '; start = ', start:5, '; pgl
  Writeln (debugger, 'finish = ', finish:5, '; teLength = ', teLength:5, '; ord4(txt) = ', ord4(t>
END;
        start := start + pglen;
        counter := counter + 1;
      END; {While start < finish}
      numpages := counter - 1;
      If not finderprint then
      BEGIN
        destrect := temprect;
        TECalText (hTE)
      END
    END;
    if debug then Writeln (debugger, 'BJDocLoop = ', PrintHdl^^.prjob.BJDocLoop, lf);
    If PrintHdl^^.prjob.BJDocLoop = BSpoolLoop
    then loop := 1
    else loop := PrintHdl^^.prjob.iCopies;
    SetPort (pointer(MyPPort));
    TextFont (NewYork);
    SetCursor (arrow);
    For counter := 1 to loop DO
    BEGIN
      HLock (txt);
      txtptr := txt^;
      For count2 := 1 to numpages DO
      BEGIN {if background printing, duplicate txt handle before starting}
        PrOpenPage (MyPPort, NIL);
        TextBox (txtptr, MyLngth[count2], pagerect, teJustLeft);
        PrClosePage (MyPPort);
        txtptr := Pointer (Ord4 (txtptr) + MyLngth[count2]);
        start := start + pglen
      END; {For count2 }
      HUnlock (txt);
    END; {For counter }
    PrCloseDoc (MyPPort);
    If PrintHdl^^.prjob.BJDocLoop = BSpoolLoop then
       PRPicFile (Printhdl, NIL, NIL, NIL, status); {omit this for spooled files.}
    if canceldialog then DisposDialog (dlogptr)
    else
    BEGIN
      disposHandle (txt);
      printing := false;
```

```
      temphdl := GetString (272);                    {change to 'Print '}
      HLock (pointer(temphdl));
      SetItem (myMenus[fileMenu], 8, temphdl^^);
      HUnlock (pointer(temphdl));
      SetPort (printport)
    END
  END
End;
```

```
{SS EditMenu}
{-------------------------------------------------------------------------------}
Procedure EditMain (theItem: integer; commandkey : boolean);
const undo = 1;
      cut = 3;
      kopy = 4;      {'Copy' is a Pascal string function}
      paste = 5;
      clear = 6;
      selectAll = 7;
      clipbored = 9; {'ClipBoard' is already used as a windowkind constant}

VAR DeskAccUp , dummy: boolean;
    Dscrap: PScrapStuff;
    off: LongInt;
    ticks: LongInt;
    tempport: grafptr;
    box: rect;
    itemhdl, hdl: handle;
    typ, io, tempstart, tempend: integer;
    tempptr: ptr;
    TextScrap: handle;
    TextLength: integer;
    Ptr2ScrapLength: ^integer;
BEGIN
{Since the Edit menu does so much, it has been broken up into a separate procedure.
 It does not yet support undo, but does support Cutting, Copying and Pasting between
 the Desk Scrap and the TextEdit Scrap.}
  DeskAccUp := false;
  IF theItem < selectAll then DeskAccUp := SystemEdit(theItem-1);
  If ((theItem in [undo,cut,kopy]) OR DeskAccUp) AND (scrapwind <> NIL) then
  Begin    {invalidate clipboard}
    GetPort (tempport);
    SetPort (scrapwind);
    Invalrect (scrapwind^.portrect);
    SetPort (tempport)
  End;
  if theItem in [cut, kopy] then
  BEGIN
    tempend := hTE^^.selend;
    tempstart := hte^^.selstart
  END;
  if (theItem > Clear) OR NOT DeskAccUp then
  BEGIN
    if debug then Writeln (debugger, 'not system edit', lf);
    { Delay so menu title will stay lit a little only if Command key }
    { equivalent was typed. }
    If commandkey then
    BEGIN
      ticks := TickCount + 10;
      REPEAT UNTIL ticks <= TickCount
    END;
    {** see if enough memory exists for move}
    CASE theItem OF
    undo:  ; { no Undo/Z in this example }
    cut:  TECut(hTE);                    { Cut/X }
    kopy:  TECopy(hTE);                  { Copy/C }
    paste:  BEGIN                        { Paste/V }
          DScrap := InfoScrap;
          If DScrap^.scrapState <> LastState then
          BEGIN
            LastState := DScrap^.scrapState;
            hdl := NewHandle (0);
            io :=GetScrap (hdl, 'TEXT', off);
            if debug then Writeln (debugger, 'io = ', io);
            If io > 0 then
            BEGIN
              TextScrap := Pointer (GlobalValue (TEScrpHandle));
```

```
            SetHandleSize (TextScrap, io);
            Ptr2ScrapLength := Pointer (GlobalAddr (TEScrpLength));
            Ptr2ScrapLength^ := io;
            Hlock (hdl);
            Hlock (TextScrap);
            BlockMove (hdl^, TextScrap^, io);
            HUnlock (hdl);
            HUnlock (TextScrap)
          END;
          DisposHandle (hdl)
        END;
        TEPaste(hTE);
      END;
  clear:  TEDelete(hTE);                    { Clear }
  selectall:  TeSetSelect(0,65535,hTE);     { Select All/A }
  clipbored:  ToggleScrap                   { Show, Hide Clipboard }
  END;    { of item case }
  If theItem in [cut,kopy] then
  BEGIN
    io := ZeroScrap;
    If debug then Writeln (debugger, 'zero scrap err =', io, lf);
    TextScrap := Pointer (GlobalValue (TEScrpHandle));
    TextLength := GetHandleSize (TextScrap);
    if debug then
Writeln (debugger, 'TextScrap @',ord4(textscrap^),'; TextLength = ',textlength,lf);
    Hlock (TextScrap);
    io := PutScrap (TextLength, 'TEXT', TextScrap^);
    If debug then Writeln (debugger, 'put scrap err =', io, lf);
    HUnlock (TextScrap)
  END;
  If theItem in [cut,clear,paste] then Windowdata^^.changed := True;
  If (theItem in [cut..clear]) then ScrollText (TRUE)
 END {not systemedit}
END;  { of editMain }
```

```
{SS Command }
{-------------------------------------------------------------------------------}
PROCEDURE MyDisable;
  const newitem = 1;
        openitem = 2;
        closeitem = 3;
        saveitem = 4;
        saveasitem = 5;
        revertitem = 6;
        pagesetupitem = 7;
        printitem = 8;
        quititem = 9;

        undoitem = 1;
        cutitem = 3;
        copyitem = 4;
        pasteitem = 5;
        clearitem = 6;
        selectallitem = 7;
        clipboreditem = 9;

var counter: integer;
    DScrap: PScrapStuff;
    temppeek: windowpeek;
    stycount: styleiten;

{-------------------------------------------------------------------------------}
  PROCEDURE KillFE (fileitems, edititems : edset);
  var counter: integer;
  BEGIN
{This guy disables the items in the File and Edit menus.  This approach has a real
 disadvantage:  If an entire menu should be disabled at some given time, there is
 no convenient way to do a DrawMenuBar here to disable the item in the bar itself.}
    If debug then
    Begin
      DebugInProc (proc, 'KillFE', @KillFE);
      Write (debugger, 'file: ');
      For counter := newitem to quititem Do
       IF counter in fileitems THEN Write (debugger, counter:2, ',');
      Write (debugger, ';  edit: ');
      For counter := undoitem to clipboreditem Do
       IF counter in edititems THEN Write (debugger, counter:2, ',');
      Writeln (debugger, 1f)
    End;
    For counter := 1 to 9 Do
    BEGIN
      If counter in fileitems then DisableItem (myMenus[FileMenu], counter);
      If counter in edititems then DisableItem (myMenus[EditMenu], counter);
    END
  END;

BEGIN
{This part goes through all of the applicable elements of the frontmost window, if any
 and from that decides what operations are allowable at this time.}
  if debug then DebugInProc (proc, 'MyDisable', @MyDisable);
  For counter := 1 to 9 DO
  BEGIN
    EnableItem (myMenus[FileMenu], counter);
    If counter in [UndoItem,CutItem..SelectAllItem,ClipboredItem]
      then EnableItem (myMenus[EditMenu], counter)
  END;
  if printing then KillFE ([PageSetupItem],[]); {page setup, if printing}
  IF Frontwindow = Nil
  THEN KillFE ([CloseItem..PrintItem],[UndoItem..SelectAllItem])
  ELSE
  BEGIN
    Mypeek := Pointer (FrontWindow);
```

```
    Case Mypeek^.windowkind of
    MyDocument:  BEGIN
          KillFE ([], [UndoItem]);
          If NOT WindowData^^.titled THEN KillFE ([SaveItem,RevertItem], []);
          If NOT WindowData^^.changed THEN KillFE ([SaveItem,RevertItem], []);
          If hTE^^.teLength = 0 THEN
            KillFE ([SaveItem,SaveAsItem,PageSetupItem,PrintItem], [SelectAllItem]);
          If hTE^^.selstart = hTE^^.selend THEN
            KillFE ([], [CutItem,CopyItem,ClearItem]);
          DScrap := InfoScrap;
          If DScrap^.scrapSize = 0 then KillFE ([], [PasteItem]);
        END;
    Clipboard,FreeMemory:  KillFE ([SaveItem..PrintItem], [UndoItem, CutItem..SelectAllItem]);
      OTHERWISE KillFE ([SaveItem..PrintItem], [SelectAllItem]) {system window}
    END {Case}
  END;
  if printing then EnableItem (MyMenus[filemenu], PrintItem) {stop printing}
END;

{-------------------------------------------------------------------------------}
PROCEDURE DoCommand (commandkey: boolean);
VAR name, s, str: str255;
    bstr: string[5];
    dummy: size;
    err : boolean;
    num, refnum, theMenu, theItem: integer;
    tempPeek: WindowPeek;
    mresult, ticks: longint;
    dipeek: DialogPeek;
    box: rect;
    itemhdl: handle;
    typ: integer;
    PrinterErrPtr : ^Integer;
BEGIN
{This handles the actions that are initiated through the Menu Manager}
  if debug then DebugInProc (proc, 'DoCommand', @DoCommand);
  MyDisable;
  If Commandkey
  then mResult := MenuKey(theChar)
  else mResult := MenuSelect (myEvent.where);
  theMenu := HiWord(mResult); theItem := LoWord(mResult);
  CASE theMenu OF
    appleMenu:  {enough memory to allow desk accessory to open}
    BEGIN
      IF theItem = 1
      THEN AboutMyEditor
      ELSE
      BEGIN
        GetItem(myMenus[appleMenu], theItem, name);
        refNum := OpenDeskAcc(name)
      END
    END;
    FileMenu:
    BEGIN
      If FrontWindow <> Nil then
       If MyPeek^.WindowKind = MyDocument then
         if windowdata^^.titled
         then GetWTitle (FrontWindow, str)
         else str := '';
      Case TheItem of
        1:  OpenAWindow;                { New }
        2:  MyGetFile;                  { Open  }
        3:  CloseAWindow;               { Close }
        4:  err :=                      { Save }
              WriteFile (windowdata^^.FileVolume, str);
        5:  err := MyPutFile (str);     { Save As  }
        6:  BEGIN                       { Revert to Saved }
              If CautionAlert(257, NIL)=OK then
```

```
              err := ReadFile (windowdata^^.FileVolume, str);
            ScrollText (FALSE)  {which is the user interfacy thing to do?
                                 display the top of the file, or display
                                 the position in the file the user was looking @
                                 when he said revert.  Should I also maintain the
                                 flashing caret position?}
          END;
       7: If PrStlDialog (PrintHdl)   { Page Setup  }
            then ;
            {eventually, store info in document resource fork}
       8: if not printing             { Print }
          then Printflag := true
          else
          BEGIN
            PrinterErrPtr := Pointer(pPrGlobals);
            PrinterErrPtr^ := IprAbort
          END;
       9: doneFlag := TRUE;           { Quit }
     END
   END;
  EditMenu: EditMain (theItem, commandkey);

{SIFC BUG > -1}
  100:
    Case theItem OF
    1:  ToggleFree;
    2:  dummy := MaxMem (dummy);
{SIFC BUG = 1}
    3:  BEGIN
          debug := not debug;
          CheckItem (MyMenus[DebugMenu], 3, debug)
        END
{SENDC}
    END  { of debug }
{SENDC}

  END;    { of menu case }
  HiliteMenu(0)
END;  { of DoCommand }
```

```
{-------------------------------------------------------------------------------}
PROCEDURE DrawWindow;
VAR tempPort : GrafPtr;
    tempscrap: handle;
    scraplength, off: longint;
    temprect, rectToErase: rect;
    str: str255;
    tempPeek: WindowPeek;
    whichwindow: windowptr;
    temphTE: TEHandle;
    tempdata: mydatahandle;
BEGIN
{ Draws the content region of the given window, after erasing whatever
  was there before. }
  if debug then DebugInProc (proc, 'DrawWindow', @DrawWindow);
  WhichWindow := Pointer (MyEvent.message);
  BeginUpdate(WhichWindow);
  GetPort (tempPort);
  SetPort (WhichWindow);
  tempPeek := Pointer (WhichWindow);
  Case tempPeek^.windowkind of
  MyDocument :
    Begin
      temprect := WhichWindow^.portrect;
      tempData := Pointer (GetWRefCon (WhichWindow));
      temphTE := tempData^^.TERecord;
      If tempPeek^.hilited then temprect.top := temprect.bottom - 15;
      temprect.left := temprect.right - 15;
      ClipRect (temprect);
      DrawGrowIcon(WhichWindow);
      Cliprect (WhichWindow^.portrect);
      DrawControls (WhichWindow);
      {this only erases the window past the end of text, if any}
      with temphTE^^ DO
      If nlines - topline < (viewrect.bottom - viewrect.top + lineheight)
       DIV lineheight then
      BEGIN
        rectToErase := viewrect;
        rectToErase.top := (nlines - topline) * lineheight;
        EraseRect (rectToErase)
      END;
      TEUpdate(WhichWindow^.visRgn^^.rgnBBox, temphTE)
    End;
  ClipBoard :
  BEGIN
    tempscrap := NewHandle (0);
    ScrapLength := GetScrap (tempscrap, 'TEXT', off);
    EraseRect (WhichWindow^.portrect);
    temprect := Whichwindow^.portrect;
    temprect.left := temprect.left + 4;
    temprect.right := temprect.right-15;
    If ScrapLength > 0 THEN
    BEGIN
      HLock (tempScrap);
      Textbox (tempscrap^, scrapLength, temprect, teJustLeft);
      HUnlock (tempScrap)
    End;
    DisposHandle (tempscrap);
    temprect := WhichWindow^.portrect;
    temprect.left := temprect.right - 15;
    ClipRect (temprect);
    DrawGrowIcon (WhichWindow);
    Cliprect (whichwindow^.portrect)
  END;
{SIFC BUG > -1}
  FreeMemory: BEGIN
        EraseRect(WhichWindow^.portRect);
```

```
        MoveTo (5, 12);
        NumToString (FreeMem, str);
        DrawString (str)
      End;
{SENDC}
  END; {Case}
  SetPort (tempPort);
  EndUpdate(WhichWindow)
END;    { of DrawWindow }
```

```
{SS CONTROL}
{-------------------------------------------------------------------------------}
PROCEDURE ScrollBits;
VAR oldvert: INTEGER;
BEGIN
{If the visible information has changed, scroll the window here.}
  if debug then DebugInProc (proc, 'ScrollBits', @ScrollBits);
  oldvert := topline;
  topline := GetCtlValue(vScroll);
  TEScroll (0, (oldvert - topline)*hTE^^.lineheight, hTE)
END;

{-------------------------------------------------------------------------------}
PROCEDURE ScrollUp(theControl: ControlHandle; partCode: INTEGER);
BEGIN
{This function is called by TrackControl in the Up button}
  if debug then DebugInProc (proc, 'ScrollUp', @ScrollUp);
  IF partCode = inUpButton THEN
  BEGIN
    SetCtlValue(theControl, GetCtlValue(theControl)-1); {VScroll}
    ScrollBits
  END
END;

{-------------------------------------------------------------------------------}
PROCEDURE ScrollDown(theControl: ControlHandle; partCode: INTEGER);
BEGIN
{This function is called by TrackControl in the Down button}
  if debug then DebugInProc (proc, 'ScrollDown', @ScrollDown);
  IF partCode = inDownButton THEN
  BEGIN
    SetCtlValue(theControl, GetCtlValue(theControl) + 1); {VScroll}
    ScrollBits
  END
END;

{-------------------------------------------------------------------------------}
PROCEDURE PageScroll(which: INTEGER);
VAR myPt: Point;
    amount: Integer;
BEGIN
{This function is called by TrackControl in the Grey part of the scrollbar}
  if debug then DebugInProc (proc, 'PageScroll', @PageScroll);
  if which = InPageUp
  then amount := -1
  else amount := 1;
  REPEAT
    GetMouse(myPt);
    IF TestControl(VScroll,myPt) = which THEN
       BEGIN
         With HTE^^.viewrect DO
           SetCtlValue(VScroll, GetCtlValue(VScroll) + amount *
            (bottom - top) DIV hTE^^.lineheight);
         ScrollBits
       END
  UNTIL NOT StillDown;
END;

{-------------------------------------------------------------------------------}
PROCEDURE MyControls;
Var t, code, whichpart: integer;
    AControl: ControlHandle;
BEGIN  {controls}
{This routine handles the scrollbar}
  if debug then DebugInProc (proc, 'MyControls', @MyControls);
  whichPart := FindControl (MyPoint, MyWindow, AControl);
If debug THEN Writeln (debugger, 'whichpart = ', whichpart, lf);
```

```
If debug THEN Writeln (debugger, 'ord( AControl = ', Ord4 ( AControl), 1f);
  {adjust scrollbar range}
  If AControl <> NIL THEN
  BEGIN
    VScroll := AControl;
    Case whichpart of
    inUpButton: t := TrackControl (VScroll, MyPoint, @scrollUp);
    inDownButton: t := TrackControl (VScroll, MyPoint, @scrollDown);
    inPageUP: PageScroll (whichpart);
    inPageDown: PageScroll (whichpart);
    inThumb:
      BEGIN
        t := TrackControl (VScroll, MyPoint, NIL);
        ScrollBits
      END
    END {Case MyControl}
  END {AControl <> NIL}
END; {controls}
```

```
{SS Initial }
{-------------------------------------------------------------------------------}
PROCEDURE SetUp;
VAR counter, vRefNum : INTEGER;
    DScrap : PScrapStuff;
    hdl, hAppparms : handle;
    off : longint;
    apName : Str255;
    NameHdl : Handle;
    strhdl : StringHandle;
    dummyrect : rect;
    tempptr : pAppParms;
    dummy : boolean;
    FinderFile : integer;
    myport : GrafPtr;
BEGIN
{Initialization for a variety of things is done here.  This code is 'discarded'
 after it is executed by an UnLoadSeg.  Another good way of initializing a large
 number of variables would be to create a custom resource which contains initial
 values for all globals.  Then, if the globals are fields in a handle, a single
 'GetResource' would initialize all fields.}

{SIFC BUG = 1}   {This code is only included for external terminal debugging}
  debug := false;{if you want debugging on as soon as the program starts, set it here}
  lf := chr(10); {At present, information written to the external terminal needs
                  its own linefeed.}
  Rewrite (debugger, '.BOUT'); {the serial port not used for downloading from Lisa}
{SENDC}

  if debug then
  BEGIN
    Writeln (debugger, lf, lf);
    DebugInProc (proc, 'SetUp', @Setup)
  END;

  {The program only executes the code when it is first run, but it could have gotten
   here in two ways.  The user may have opened the application or one of its
   documents, or the user may have chosen to print a document.  In any case, some
   common initialization is needed.}

  SetGrowZone (@MyGrowZone);  {just in case something goes wrong..}
  InitGraf(@thePort);         {I need QuickDraw}
  InitFonts;                  {I need fonts}
  InitWindows;                {I need windows}
  FlushEvents(everyEvent,0);  {start with a clean slate}
  TEInit;                     {I need TextEdit}
  InitDialogs(NIL);           {and I need dialogs, even when printing from Finder}

  NameHdl := NewHandle (4000000); {force MemMgr to allocate all 'grow' to app.}

  PrintHdl := Pointer (NewHandle (120));
  PrOpen;
  PrintDefault (PrintHdl);
  getAppParms(apName,vRefNum,hAppParms);
  HLock(hAppParms);
  {** sometime, get file info for apName, to use folder info as appropriate}
  tempptr := Pointer(hAppParms^);
  iBeamHdl := GetCursor(1);
  HNoPurge (Pointer(iBeamHdl));
  watchHdl := GetCursor(4);
  HNoPurge (Pointer(watchHdl));
  numfiles := tempptr^.count;
  if debug then Writeln (debugger, 'numfiles=',numfiles,lf);
  finderprint := (tempptr^.message = 1);
  IF finderprint {User selected 'print' from the Finder} THEN
  BEGIN
    setport (thePort);
```

```
  dummyrect := screenbits.bounds;
  dummyrect.bottom := dummyrect.top + 16;
  InsetRect (dummyrect,10,2);
  New (myPort);
  OpenPort (myPort);
  TextBox (pointer(ord4(@apName)+1),ord4(Length(apName)),dummyrect,teJustCenter);
  For FinderFile := 1 to numfiles Do
   With tempptr^ DO
  BEGIN
    If ftype = 'TEXT' then
    BEGIN
      SetRect (dummyrect, 0,0,100,100);
      SetPort (myPort);                    {to allow text measure in TeCalText}
      hTE := TENew(dummyrect, dummyrect);
      dummy := ReadFile (vRefNum, fName); {assume that page setup is read in as well}
      Unloadseg (@ReadFile);
      MyPrint(FinderFile, fName);
      SetCursor (watchhdl^^);
      TEDispose (hTE); {dispose of text edit stuff}
      tempptr := Pointer (Ord4 (tempptr) + length (fName) + 10 -
       length (fName) MOD 2)
    END
    {ELSE clear the proper bytes in the appParms handle?}
  END;
  hTE := NIL;
  ClosePort (myPort)
END
ELSE
BEGIN
  InitMenus;   { initialize Menu Manager }
  myMenus[appleMenu] := GetMenu(appleMenu);
  myMenus[appleMenu]^^.menudata[1] := CHR(Applesymbol);
  AddResMenu(myMenus[1],'DRVR');    { desk accessories }
  For counter := FileMenu to EditMenu DO myMenus[counter] := GetMenu(counter);
{SIFC BUG > -1}
  myMenus[DebugMenu] := GetMenu(100);       { temporary debug menu }
{SENDC}
{SIFC BUG = 1}
  extdebughdl := GetString (261);
  HLock (Pointer(extdebughdl));
  AppendMenu (myMenus[DebugMenu], extdebughdl^^);
  HUnlock (Pointer(extdebughdl));
  ReleaseResource (Pointer(extdebughdl));
  CheckItem (MyMenus[DebugMenu], 3, debug);
{SENDC}
  FOR counter:=1 TO lastMenu DO InsertMenu(myMenus[counter],0);
  DrawMenuBar;
  dragRect := screenbits.bounds;
  dragrect.top := dragrect.top + 20; {leave room for menu bar}
  growRect := dragRect;
  InsetRect (dragrect, 4, 4); {leave some of dragged rectangle on screen}
  growrect.left := {replace this with the max font width + constant} 80;
  growrect.top := 80 {18 + 16*3 + slop?};
  doneFlag := FALSE;
  printflag := false;
  printing := false;
  windownum := 1;
  windowpos := 0;
  MyFileTypes[0] := 'TEXT';
  Laststate := 0; {eventually, init laststate to scrapstate - 1?}
  For counter := 1 to numfiles Do
   With tempptr^ DO
  BEGIN
    If ftype = 'TEXT' then
    BEGIN
      MakeAWindow (fName, TRUE); {**could async open while this is going on}
      if counter < numfiles then DialogueDeactivate;
      If Not ReadFile (vRefNum, fName) then
```

```
        BEGIN
          TEDispose (hTE);
          hTE := NIL;
          DisposHandle (Pointer (WindowData));
          DisposeWindow (MyWindow)
        END;
        tempptr := Pointer(Ord4 (tempptr) + length(fName) + 10 - length(fName) MOD 2)
      END
    END;
    If Frontwindow = NIL then OpenaWindow;
    {if something 'TEXT' is in deskscrap then allow paste}
    DScrap := InfoScrap;
    LastState := DScrap^.scrapState;
    If DScrap^.scrapsize > 0 then LastState := LastState - 1;
    {what about when scrapsize is too big?}
    Scrapwind := NIL;
  {SIFC BUG > -1}
    Freewind := NIL
  {SENDC}
  END;
  Hunlock (happParms)
END;    { of SetUp}
```

```
{SS          }
{-------------------------------------------------------------------------------}
PROCEDURE CursorAdjust;
VAR mousePt:  Point;
    tempport: grafptr;
    temppeek: Windowpeek;
BEGIN
{ Take care of application tasks which should be executed when the machine has
  nothing else to do, like changing the cursor from an arrow to an I-Beam when it
  is over text that can be edited. }
{SIFC BUG >-1}
{ If the amount of free memory is being displayed in its own window, and if it has
  changed, then create an update event so that the correct value will be displayed.}
  If (FreeWind <> NIL)
  and (FreeMem <> OldMem) then
  BEGIN
    OldMem := FreeMem;
    GetPort (tempport);
    SetPort (FreeWind);
    InvalRect (FreeWind^.portrect);
    SetPort (tempport)
  END;
{SENDC}
  GetMouse(mousePt); {where the cursor is, currently (local to the topmost window)}
  If hTE <> NIL {if text edit is currently active, (document window is topmost)}
  THEN
  BEGIN
    TEIdle (hTE);
    IF (PtInRect(mousePt, hTE^^.viewrect)) {In the text edit viewrect area,}
    THEN SetCursor(iBeamHdl^^)             { make the cursor an I-beam.}
    ELSE SetCursor(arrow)
  END
  ELSE
  BEGIN
    {let desk accessories set their own?}
    temppeek := pointer(FrontWindow);
    If temppeek = NIL then SetCursor (arrow)
    else if temppeek^.windowkind > 1 then SetCursor (Arrow)
  END
END;

{-------------------------------------------------------------------------------}
FUNCTION MyGrowZone;
BEGIN
{This function is called by the memory manager whenever more memory is requested than
 available.  The only time you'll see it in this program is when it initally runs
 (which is normal) and when it is not checking memory availability when it should.
 Your program should not rely on resolving memory problems here, because it could be
 called by the ROM, where, at present, insufficient memory cases are not always
 handled gracefully.}
  if GZCritical then
  BEGIN
    if debug then Writeln (debugger, 'myGrow cbneeded = ', cbneeded, lf);
    {Make all data stuctures, including user data, that can be safely released,
     purgable.  If the user has data in memory that has not yet been saved, and if
     you were not expecting this routine to be called, then the call came from ROM
     and is important to give the user the chance to save their work.  Even if
     their data is successfully saved, it is likely that the program will have to
     restart or quit to the Finder.}
  END;
  MyGrowZone := 0 {for now, the memory requests fails unconditionally}
END;

{-------------------------------------------------------------------------------}
PROCEDURE MainEventLoop;
Var code: integer;                {the type of mousedown event}
    dummy: boolean;
```

```
    str : str255;
    tempport : Grafptr;
BEGIN
{This event loop handles most of the communications between this program and events
taking place in the outside world.  This procedure is also called as the printer
idle procedure so that the program appears to be doing background printing.}

  if printing then
  BEGIN
    getport (tempport);
    setport (printport)
  END;
  REPEAT
    CursorAdjust;
    SystemTask;
    if printflag then
    BEGIN
      GetWTitle (MyWindow, str);
      Myprint(1, str) {number of files to print, what to call it}
    END;
    dummy := GetNextEvent(everyEvent,myEvent);
    CASE myEvent.what OF
    mouseDown:
      BEGIN
        code := FindWindow(myEvent.where,tempWindow);

        CASE code OF
          inMenuBar: DoCommand(FALSE);
          inSysWindow: InSystemWindow;
          inDrag: DragWindow(tempWindow,myEvent.where,dragRect);
          inGoAway: IF TrackGoAway(tempWindow,myEvent.where) THEN CloseAWindow;
          inGrow: If Mypeek^.windowkind in [MyDocument,Clipboard] then GrowWnd;

          inContent:
            BEGIN
              IF tempWindow <> FrontWindow
              THEN  SelectWindow (tempWindow)
              ELSE
               IF hTE <> NIL THEN
              BEGIN
                MyPoint := MyEvent.where;
                GlobalToLocal (MyPoint);
                IF PtInRect (MyPoint, hTE^^.viewrect)
                THEN
                BEGIN
                  If debug THEN Writeln (debugger, 'point in HTE viewrect', lf);
                  IF (BitAnd (myEvent.modifiers, ShiftKey) <> 0 ) { Shift key pressed }
                  THEN TEClick (MyPoint, TRUE, hTE)
                  ELSE TEClick (MyPoint, FALSE, hTE);
                END
                ELSE MyControls
              END {hTE <> NIL}
            END {in Content}
          END { of code case }
        END; { of mouseDown }

      keyDown, autoKey:
        BEGIN
          theChar := CHR(myEvent.message MOD 256);     { Mac characters use 8 bits }
          IF BitAnd(myEvent.modifiers,CmdKey) <> 0     { Command key pressed }
          THEN DoCommand(TRUE)
          ELSE IF hTE <> NIL THEN
          BEGIN
            TEKey(theChar,hTE);
            windowdata^^.changed := true;
            ScrollText (TRUE);
          END
        END;     { of keyDown }
```

```
      activateEvt: MyActivate;
      updateEvt: DrawWindow;
      nullEvent: If doneflag AND (FrontWindow <> NIL) Then CloseAWindow;
    END;    { of event case }
    UnloadSeg (@InSystemWindow); {segment Utilities}
    UnloadSeg (@ReadFile);       {segment ReadFile}
    UnloadSeg (@WriteFile);      {segment WritFile}
    UnloadSeg (@AboutMyEditor);  {segment AboutMyPgm}
    UnloadSeg (@MyDisable);      {segment DoCommand}
    UnloadSeg (@Scrollbits);     {segment Control}
    if not printing then UnloadSeg (@MyPrint);
  UNTIL (doneFlag AND (FrontWindow = NIL)) or Printing;
  if doneFlag AND (FrontWindow = NIL) then clearmenubar;  {prevent the user from
   doing anything until printing is through}
  if printing then
  BEGIN
    getport (printport);
    setport (tempport)
  END
END;

BEGIN    { main program }
{Please don't look at this program as the the last word in example programming, and
be very cautious about porting some portion of this program over to your own code.}
  SetUp;
  UnloadSeg (@Setup);
  if not finderprint then MainEventLoop;
  SetCursor (watchHdl^^);
  PrClose
END.
```

```
;
;--------------------------------------------------------------------------
;    These routines provides the Pascal programmer with assembly-like constructs
;    that Pascal can not perform easily.
;
;    FUNCTION GlobalAddr
;
;    Given a low memory location name (a constant in the example program File),
;    the correct address for the routine is returned.  This function should be
;    declared in the first part of your program as :
;
;         FUNCTION GetGlobalAddr (GlobalConst : Integer): Ptr; External;
;
;    Suppose a pointer in your program is declared as:
;
;         VAR somePtr : Ptr;
;
;    To store a value in a global location, do the following:
;
;         somePtr := GetGlobalAddr (AGlobal);
;         somePtr^ := NewGlobalValue;
;
;    Note that since Ptr is defined as ^signedbyte, this only writes a single byte.
;
;    To write to some other data type, you must declare a pointer to that
;    data type, and then use the Pointer function to equate the result of this
;    function to that data type.  For example, to write a long word, declare a new
;    variable:
;
;         VAR bignumptr = ^LongInt;
;
;    Then, equate the variable to the function, and assign a value to it to perform
;    the write:
;
;         bignumptr := Pointer (GetGlobalAddr (AGlobal));
;         bignumptr^ := some long integer expression
;
;
;    FUNCTION GlobalValue
;
;    Given a low memory location name (a constant in the example program File),
;    the value stored at that low memory address is returned.  This function should
;    be declared in the first part of your program as :
;
;         FUNCTION GlobalValue (GlobalConst : Integer): LongInt; External;
;
;    Just assign the function result to a long integer to return that value.
;
;    To read some other data type, you must declare a pointer to that data type,
;    and then use the Pointer function to equate this result to that data type.
;
;   For example, to read a global value into your own special handle, assign:
;
;         MyHandle := Pointer(GlobalValue (AGlobal));
;
;   Note that this does not create a new or duplicate handle called MyHandle.  This
;   only provides you with a method of manipulating the existing handle contained in
;   AGlobal.
;
;  MODIFICATION HISTORY
;
;     06-Feb-84  CRC  New Today
;--------------------------------------------------------------------------
;
               .NOLIST
               .INCLUDE TlAsm/GrafTypes.Text
               .INCLUDE TlAsm/QuickMacs.Text
               .INCLUDE Tlasm/SysEqu.Text
```

```
                .INCLUDE TlAsm/ToolEqu.Text
                .INCLUDE TlAsm/ToolMacs.Text
                .LIST

;          FUNCTION GlobalAddr (GlobalConst : Integer): Ptr;
;          FUNCTION GlobalValue (GlobalConst : Integer): LongInt;

                .FUNC     GlobalAd,0
                .DEF      GlobalVa

                MOVEQ     #0,D1               ;address entry
                BRA.S     GlobStart

GlobalValue     MOVEQ     #1,D1               ;data entry

GlobStart       MOVE.L    (SP)+,A1            ;preserve return address
                MOVE.W    (SP)+,D0            ;the routine # requested
                ASL       #2,D0               ;make it into a long offset
                LEA       TableBase,A0        ;get beginning of table
                MOVE.L    0(A0,D0),A0         ;get the value out of the table
                TST       D1                  ;which entry?
                BEQ.S     @1                  ;branch if address
                MOVE.L    (A0),A0             ;dereference if data
@1              MOVE.L    A0,(SP)             ;and put value in function return
                JMP       (A1)                ;bye for now

;  These addresses were chosen because they are frequently needed by applications,
;  and are not readily available in existing globals or ToolBox calls.
;  any address can be read as data or written to as an address.
;  Additional globals will be added as they are requested.

TableBase .LONG     TEScrpLength        ;the length of the private TextEdit scrap
          .LONG     TEScrpHandle        ;the handle to the private TextEdit scrap
          .LONG     dlgFont             ;the font used inside alerts and dialogs
          .LONG     ScrVRes             ;screen vertical resolution (dots/inch)
          .LONG     ScrHRes             ;screen horizontal resolution (dots/inch)
          .LONG     doubleTime          ;double click time in 4/60's of a second
          .LONG     caretTime           ;caret blink time in 4/60's of a second
          .LONG     ANumber             ;the active alert
          .LONG     ACount              ;the alert stage level

;-------------------------------------------------------------------------------
;
;     Procedure AutoScroll;
;
;         The location of this procedure is passed to TextEdit in the clikLoop field.
;         It is called by TextEdit when the user drags a selection range outside of
;         the viewrect.  This calls the pascal procedures ScrollUp and ScrollDown
;         to cause the screen to scroll, if possible, and the selection range to be
;         extended.
;
;-------------------------------------------------------------------------------

                .PROC     AutoScroll,0

;offsets for Pascal globals

MyWindow        .EQU      -4                  ;offset for current application window
VScroll         .EQU      -20                 ;the window's vertical scroll bar

                .REF      ScrollDown
                .REF      ScrollUp

                PEA       temppoint
                _GetMouse                     ;get local mouse point to D0
                MOVE.W    temppoint,D0

;Now see if we're in the text rect.
```

```
                LEA     Condition,A0            ;a place to store the result
                CMP.W   TEViewRect+Top(A3),D0   ;Compare with Top
                BLT.S   AS_OutOfRect            ;Yep, he moved above top!
                CMP.W   TEViewRect+Bottom(A3),D0 ;is Mouse < bottom?
                BLE.S   AS_NoMove               ;no, don't scroll.

AS_OutOfRect    MOVE    SR,(A0)                 ;save top or bottom in Condition
                MOVE.L  -4(A5),-(SP)            ;global MyWindowPtr
                _SetPort                        ;set application's port
                MOVE.L  -4(A5),A0               ;global MyWindowPtr
                PEA     PortRect(A0)            ;global MyWindowPtr^.portrect
                _ClipRect                       ;set the application's clip
                MOVE.L  -20(A5),-(SP)           ;push handle for scroll, below
                MOVE    Condition,CCR           ;get back top or bottom condition
                BLT.S   AS_OffTop

;We're off the bottom. do a scroll Down.
                MOVE.W  #inDownButton,-(SP)     ;get const. req'd by ScrollDown.
                JSR     ScrollDown
                BRA.S   AS_RestorePort          ;go try again.

;We're off the top. Do a scroll Up.
AS_OffTop       MOVE.W  #inUpButton,-(SP)       ;get constant req'd by ScrollUp
AS_OT2          JSR     ScrollUp                ;go scroll the line.
AS_RestorePort  MOVE.L  TEGrafPort(A3),-(SP)
                _SetPort                        ;restore TE's port
                PEA     TEViewRect(A3)
                _ClipRect                       ;restore the clip

AS_NoMove       MOVEQ   #-1,D0                  ;return non-zero!
                RTS

Condition
temppoint       .LONG   0
                .END
```

```
*  FileResDef -- Resource input for sample application named File
*                Written by Macintosh User Education

Example/File.Rsrc

Type CARY = STR
   ,0
File  Version 1.0  February 28, 1984

Type FREF = HEXA
   ,128(32)
4150504C
0000
00

Type FREF = HEXA
   ,129(32)
54455854
0001
00

Type BNDL = HEXA
   ,128
434152590000
0001
49434E230001
0000 0080
0001 0081
465245460001
0000 0080
0001 0081

Type ICN# = HEXA
 ,128(32)
00000000
00000000
00000000
00020000
00050000
00088038
00104044
00202082
00401102
00800A82
01000544
02000AA8
04001550
08002AA0
10005540
2000AAA0
40001510
80010A08
40000410
20030820
1003A040
08038080
04000100
02000200
01000400
00800800
00401000
00202000
00104000
00088000
00050000
00020000
00000000
00000000
```

```
00000000
00020000
00070000
000F8038
001FC07C
003FE0FE
007FF1FE
00FFFBFE
01FFFFFC
03FFFFF8
07FFFFF0
0FFFFFE0
1FFFFFC0
3FFFFFE0
7FFFFFF0
FFFFFFF8
7FFFFFF0
3FFFFFE0
1FFFFFC0
0FFFFF80
07FFFF00
03FFFE00
01FFFC00
00FFF800
007FF000
003FE000
001FC000
000F8000
00070000
00020000

,129 (32)
0FFFF800
08000400
08000600
08000500
08000480
08000438
08000444
08000682
08000102
08000282
08000544
08000AA8
08001550
08002AA0
08005550
0800AA90
08001510
08010A10
08000410
08030810
0803A010
08038010
08000010
08000010
08000010
08000010
08000010
08000010
08000010
08000010
0FFFFFF0
0FFFF800
0FFFFC00
0FFFFE00
0FFFFF00
```

```
0FFFFF80
0FFFFFF8
0FFFFFFC
0FFFFFFE
0FFFFFFE
0FFFFFFE
0FFFFFFC
0FFFFFF8
0FFFFFF0
0FFFFFE0
0FFFFFF0
0FFFFFF0
0FFFFFF0
0FFFFFF0
0FFFFFF0
0FFFFFF0
0FFFFFF0
0FFFFFF0
0FFFFFF0
0FFFFFF0
0FFFFFF0
0FFFFFF0
0FFFFFF0
0FFFFFF0
0FFFFFF0
0FFFFFF0
0FFFFFF0
0FFFFFF0


Type MENU
  ,1 (4)
  @
    About File
    (-

  ,2 (4)
  File
    New
    Open
    Close
    Save
    Save As
    Revert to Saved
    Page Setup
    Print
    Quit

  ,3 (4)
  Edit
    Undo/Z
    (-
    Cut/X
    Copy/C
    Paste/V
    Clear
    Select All/A
    (-
    Show Clipboard

  ,100 (4)
  Debug
    Show FreeMem
    Compact Memory

* document window
Type WIND
```

```
  ,256 (4)
x
  50 40 158 472
  Visible NoGoAway
  2
  0

  ,257 (32)
Clipboard
  262 4 337 446
  Visible GoAway
  0
  0

  ,258 (32)
FreeMem
  320 442 341 511
  Visible NoGoAway
  0
  0

* vertical scroll bar
Type CNTL
  ,256 (4)
x
  -1 395 236 411
  invisible
  16
  0
  0 0 0

Type DITL
  ,256 (32)
  4
    BtnItem Enabled
    65 13 85 83
Yes

    BtnItem Enabled
    95 300 115 370
Cancel

    BtnItem Enabled
    95 13 115 83
No

    StatText Disabled
    8 60 60 370
Do you want to save changes made to ^0^1?

  ,257 (32)
  3

    BtnItem Enabled
    90 267 110 337
OK

    StatText Disabled
    10 60 70 350
An error occured while ^0 the disk.  The file ^1 was not ^2.

    StatText Disabled
    90 10 110 260
^3

  ,258 (32)
  3
```

```
    BtnItem Enabled
    62 300 82 370
Cancel

    StatText Disabled
    5 10 60 370
The document '^0' is being spooled to disk and printed.

    StatText Disabled
    62 10 82 270
^1 of ^2.

  ,259 (32)
  3

    BtnItem Enabled
    90 13 110 83
OK

    BtnItem Enabled
    90 267 110 337
Cancel

    StatText Disabled
    10 60 70 350
Are you sure you want to go back to the old version of this file?  You will lose any changes that

  ,260 (32)
  3

    BtnItem Enabled
    90 13 110 83
OK

    BtnItem Enabled
    90 267 110 337
Cancel

    StatText Disabled
    10 60 70 350
A file by that name is already open.  '^0' will be opened instead.

Type DLOG
* this is the 'press cancel to stop printing' dialog
  ,257 (32)
  40 66 125 446
  Visible 1 NoGoAway 0
  258

Type ALRT
* a stop alert - an error occured while reading or writing the disk
  ,256 (32)
  60 81 180 431
  257
  5555

* a caution alert - a file is changed and 'Revert to Saved' is chosen
  ,257 (32)
  60 81 180 431
  259
  CCCC

* a note alert - the file selected is already on the desktop
  ,258 (32)
  60 81 180 431
  260
  CCCC
```

```
* a caution alert - the file is being closed, but has not yet been saved
  ,259 (32)
  60 66 180 446
  256
  4444

 Type STR
  ,256 (36)
Untitled-
  ,257 (32)
Save this document as:
  ,259 (32)
File, by Cary Clark   Version 1.0    February 7, 1984
  ,260 (32)
This example was written to demonstrate the Macintosh User Interface.
  ,261 (36)
External Debugger
  ,262 (32)
Show Clipboard
  ,263 (32)
Hide Clipboard
  ,264 (32)
Show FreeMem
  ,265 (32)
Hide FreeMem
  ,266 (32)
 before quitting
  ,267 (32)
reading from
  ,268 (32)
writing to
  ,269 (32)
loaded
  ,270 (32)
saved
  ,271 (32)
 more files to go.
  ,272 (32)
Print
  ,273 (32)
Stop Printing
  ,274 (32)
Copy of
  ,275 (32)
This disk is full.
  ,276 (32)
The disk directory is full.
  ,277 (32)
This file is locked.
  ,278 (32)
The disk is locked.
  ,279 (32)
The disk is unreadable.
  ,280 (32)
ID =

Type CODE
  Example/fileL,0
```

Index to Technical Documentation /TOOLBOX/INDEX

See Also: Inside Macintosh: A Road Map
Programming Macintosh Applications in Assembly Language
The Resource Manager: A Programmer's Guide
QuickDraw: A Programmer's Guide
The Font Manager: A Programmer's Guide
The Event Manager: A Programmer's Guide
The Window Manager: A Programmer's Guide
The Control Manager: A Programmer's Guide
The Menu Manager: A Programmer's Guide
TextEdit: A Programmer's Guide
The Dialog Manager: A Programmer's Guide
The Desk Manager: A Programmer's Guide
The Scrap Manager: A Programmer's Guide
Toolbox Utilities: A Programmer's Guide
Macintosh Packages: A Programmer's Guide
The Memory Manager: A Programer's Guide
The Segment Loader: A Programmer's Guide
The File Manager: A Programmer's Guide
The Structure of a Macintosh Application
Putting Together a Macintosh Application

Modification History:	First Draft	Caroline Rose	8/5/83
	Second Draft	Caroline Rose	1Ø/5/83
	Third Draft	Caroline Rose	1/9/84
	Fourth Draft	Caroline Rose	6/5/84

ABSTRACT

This is an index to all the documentation listed under "See Also:" above, as of 6/5/84. It will be expanded and updated periodically.

INDEX

The page numbers are preceded by a two-letter designation of which manual the information is in:

AL	Programming Macintosh Applications in Assembly Language	2/27/84
CM	The Control Manager: A Programmer's Guide	5/30/84
DL	The Dialog Manager: A Programmer's Guide	11/16/83
DS	The Desk Manager: A Programmer's Guide	9/26/83
EM	The Event Manager: A Programmer's Guide	6/20/83
FL	The File Manager: A Programmer's Guide	5/21/84
FM	The Font Manager: A Programmer's Guide	2/7/84
MM	The Memory Manager: A Programmer's Guide	10/10/83
MN	The Menu Manager: A Programmer's Guide	11/1/83
PK	Macintosh Packages: A Programmer's Guide	5/7/84
PT	Putting Together a Macintosh Application	4/9/84
QD	QuickDraw: A Programmer's Guide	3/2/83
RD	Inside Macintosh: A Road Map	12/22/83
RM	The Resource Manager: A Programmer's Guide	3/8/84
SL	The Segment Loader: A Programmer's Guide	6/24/83
SM	The Scrap Manager: A Programmer's Guide	11/16/83
ST	The Structure of a Macintosh Application	2/8/84
TE	TextEdit: A Programmer's Guide	9/28/83
TU	The Toolbox Utilities: A Programmer's Guide	2/8/84
WM	The Window Manager: A Programmer's Guide	5/30/84

E

User's Guide -- an overview of FP68K ELEMS68K and their design philosophy.

Programmer's Guide -- hints on how to build the packages, and how to modify them, if necessary; details about system dependencies involving the state area. Includes register map templates.

System Interface -- how FP68K and ELEMS68K affect their execution environment.

High-Level Interface -- the SANE Pascal unit and assembly macros.

Integer Conversion Tests

Binary-Decimal Conversion Tests

IEEE Tests -- a set of test vectors designed for this style of arithmetic and distributed through the standards subcommittee

Binary-Decimal Conversions -- what is available through the SANE interface, and what FP68K provides at the low level. A sample parser and formatter from the SANE interface is shown.

P754 stuff -- papers related to the arithmetic standard.

```
FPxxx.TEXT -- source files for FP68K, except for binary-decimal
        conversions

FBxxx.TEXT -- source files for binary-decimal part of FP68K

SAxxx.TEXT -- SANE68.TEXT  -- SANE interface section
              SAIMP68.TEXT -- SANE implementation section
              SAASM68.TEXT -- SANE assembly procedures
              SAMAC68.TEXT -- EQU's and MACRO's for assembly interface

DOxxx.TEXT -- documentation using SCRIPT formatter, with
        macros in DODRIVER.TEXT

TVxxx.TEXT -- IEEE test vector files, required operations

TWxxx.TEXT -- IEEE test vector files, appendix funtions

TDxxx.TEXT -- Test vector driver program files

ITxxx.TEXT -- integer <--> extended conversion tests

IOxxx.TEXT -- binary <--> decimal conversions tests

Zyyyy.OBJ  -- executable test programs

ELxxx.TEXT -- elementary transcendental and financial functions
```

Introduction

The 68000 software floating-point packages, FP68K and ELEMS68K, appear much like simple subroutines but their interaction with the host system is somewhat more subtle. This document indicates possible trouble spots. It is intended for system implementors, rather than users of FP68K and ELEMS68K.

The following sections describe the various issues in turn.

Registers and stack used

FP68K and ELEMS68K receive all of their parameters on the stack. They save and restore all of the CPU registers across calls, except that D0 is modified by the REMAINDER operation. FP68K modifies the CPU Condition Code Register as described later.

As detailed in the "Program Notes" document, FP68K typically uses up to 41 words of stack beyond the input parameters. The only exceptions are the binary-decimal conversion and nextafter routines, which may use up to 120 words beyond the input parameters. ELEMS68K uses at most 30 words of stack for temporary storage.

Single entry point

FP68K has just one entry point -- with the label 'FP68K'. When invoked, FP68K expects the return address on the stack, followed by a one-word opcode described in the user's guide. Beyond the opcode are up to three operand addresses (depending on the operation). Note that because the operands are passed by address, they must be in memory, NOT IN THE REGISTER FILE.

If FP68K is to be invoked by a mechanism like the A-line trap, care must be taken that stack is set up properly. Depending on the system, it should be possible to execute FP68K either as a subprogram linked to an application program, or as system-provided utility.

Because of the varying number of input parameters, it is impossible to call FP68K directly from Pascal, since the number of parameters is fixed when the EXTERNAL procedure is defined. In any case programmers should use the provided Pascal interface, called SANE (Standard Apple Numeric Environment).

ELEMS68K has a similar design, but is configured as a separate package for modularity.

Exit points

Typically, FP68K exits by clearing all input operands from the stack and jumping to the return address.

However, a 'halting' mechanism is provided whereby control is transferred from FP68K to an address saved in the floating-point state area (see below). This address should refer to a subprogram in the user's code space. When the halt routine is invoked, the top of the stack is a word containing the number of bytes of parameters (including the return address) on the stack when FP68K was originally called. Beyond that word is the exact stack frame from when FP68K was originally called.

ELEMS68K has no built-in halt mechanism, though a subsidiary FP68K operation may halt.

State area

ELEMS68K maintains no static state. FP68K maintains 3 words of static state across invocations. The first word contains mode and flag bits, much like the CPU Status Register. The next long word is the user trap address. There are two important issues: where is the state area and how is it initialized?

The state area may be a fixed area in memory, as in MAC, or at a fixed offset from a register like A6, as in LISA, or in some user area if FP68K is linked as a subroutine. The state area may even be kept within FP68K itself, though this makes the code self-modifying and thus NON-REENTRANT.

In multi-process environments, care must be taken to see that different state areas are kept for the different processes (again, think of the CPU Status Register). For example, if the state area is kept in a fixed location in memory, it must be swapped each time a new process is swapped in.

The location of the state area must be known at ASSEMBLY TIME. As indicated in the programmer's guide document, the code must be set up for the particular host environment.

When a new process is started up, the state area must be initialized. Fortunately, this is easy. Just clear to 0 the first word of the state area (i.e. the mode and flag word).

CPU Condition Code Register

The Comparison operation leaves the CCR in a well-defined state. After Comparison, the CCR is set for a conditional branch, although the flags are used in a way different from the integer CPU comparisons; see the "User's Guide" for details.

CPU Register D0

The Remainder operation leaves the low-order integer quotient (between -127 and +127) in D0.W. The high half of D0.L is undefined. This intrusion into the register file is extremely valuable in argument reduction -- the principal use of Remainder. The state of D0 after an invalid remainder is

undefined.

SANE

There is a SANE (Standard Apple Numeric Environment) library of utility functions based on FP68K, as well as a corresponding Elems library based on ELEMS68K. These libraries are supported on Apple III Pascal systems as well. The library provides access to the package from (Lisa) Pascal. Aside from support of basic arithmetic and elementary functions, the utilities manipulate the modes and flags and provide ASCII <--> floating-point conversions. All applications software should use this package because of its high degree of portability.

Assembly language programmers will invoke FP68K and ELEMS68K directly but will depend on some library for routines to convert between ASCII strings and the canonical decimal format which FP68K recognizes. A set of mnemonic MACROS has been provide to expedite assembly coding.

Compiling Pascal programs

A Pascal program which exploits the SANE and Elems interfaces must include lines such as

```
uses {$U <some volume>:SANE.OBJ} SANE;
uses {$U <some volume>:ELEMS.OBJ} Elems;
```

in order to gain access to the types and procedures defined there. Then the program must be linked with SANE.OBJ and ELEMS.OBJ (the Pascal parts of the interface), as well as SANEASM.OBJ and ELEMSASM.OBJ (the assembly language parts of the interface).

Pascal procedures

Programmers should consult the INTERFACE section of the SANE and Elems interfaces (files SANE.TEXT and ELEMS.TEXT) in the following pages. This interface reflects the architecture discussed in the "User's Guide". It is two-address, with the destination operand in the extended format except for format conversions conversions.

Macros

A set of macros provides direct contact with the arithmetic package, using the interface described in the "User's Guide". The macros take care of the opcode and the JSR, but the programmer must explicity push the required argument addresses. The macros do not take effective address arguments and push them itself because of the problems that arise if the destination operand is given as an offset from SP (which changes when the first operand address is pushed). The macros are listed after the Pascal interface.

Sample program

The test programs ITxxx.TEXT, IOxxx.TEXT, and TDFP.TEXT provide a nontrivial view of how to use the Pascal interface to FP68K and ELEMS68K.

```
{^he ''SANE Interface'' }
{^fo '28 December 1982'Page %'Apple Confidential' }
{$C Copyright Apple Computer, 1982 }
{MacIntosh version.}

UNIT Sane;

    INTERFACE

        CONST

            SIGDIGLEN = 20;      { Maximum length of SigDig. }

            DECSTRLEN = 80;      { Maximum length of DecStr. }

        TYPE

            {-------------------------------------------------------------
            ** Numeric types.
            -------------------------------------------------------------}

                Single   = array [0..1] of integer;
                Double   = array [0..3] of integer;
                Comp     = array [0..3] of integer;
                Extended = array [0..4] of integer;

            {-------------------------------------------------------------
            ** Decimal string type and intermediate decimal type,
            ** representing the value:
            **      (-1)^sgn * 10^exp * dig
            -------------------------------------------------------------}

                SigDig   = string [SIGDIGLEN];
                DecStr   = string [DECSTRLEN];
                Decimal  = record
                               sgn : 0..1;    { Sign (0 for pos, 1 for neg). }
                               exp : integer; { Exponent. }
                               sig : SigDig   { String of significant digits. }
                           end;

{^ne 16 }
            {-------------------------------------------------------------
            ** Modes, flags, and selections.
            ** NOTE: the values of the style element of the DecForm record
            ** have different names from the PCS version to avoid name
            ** conflicts.
            -------------------------------------------------------------}

                Environ   = integer;
                RoundDir  = (TONEAREST, UPWARD, DOWNWARD, TOWARDZERO);
                RelOp     = (GT, LT, GL, EQ, GE, LE, GEL, UNORD);
                              { >   <  <>  =   >=  <=  <=> }
                Exception = (INVALID, UNDERFLOW, OVERFLOW, DIVBYZERO,
                                                                INEXACT);
```

```
            NumClass  = (SNAN, QNAN, INFINITE, ZERO, NORMAL, DENORMAL);
            DecForm   = record
                            style  : (FloatDecimal, FixedDecimal);
                            digits : integer
                        end;

{^ne 35 }
        {-----------------------------------------------------------------
        ** Two address, extended-based arithmetic operations.
        -----------------------------------------------------------------}

        procedure AddS (x : Single;   var y : Extended);
        procedure AddD (x : Double;   var y : Extended);
        procedure AddC (x : Comp;     var y : Extended);
        procedure AddX (x : Extended; var y : Extended);
            { y := y + x }

        procedure SubS (x : Single;   var y : Extended);
        procedure SubD (x : Double;   var y : Extended);
        procedure SubC (x : Comp;     var y : Extended);
        procedure SubX (x : Extended; var y : Extended);
            { y := y - x }

        procedure MulS (x : Single;   var y : Extended);
        procedure MulD (x : Double;   var y : Extended);
        procedure MulC (x : Comp;     var y : Extended);
        procedure MulX (x : Extended; var y : Extended);
            { y := y * x }

        procedure DivS (x : Single;   var y : Extended);
        procedure DivD (x : Double;   var y : Extended);
        procedure DivC (x : Comp;     var y : Extended);
        procedure DivX (x : Extended; var y : Extended);
            { y := y / x }

        function  CmpX (x : Extended; r : RelOp;
                                      y : Extended) : boolean;
            { x r y }

        function  RelX (x, y : Extended) : RelOp;
            { x RelX y, where RelX in [GT, LT, EQ, UNORD] }

{^ne 18 }
        {-----------------------------------------------------------------
        ** Conversions between Extended and the other numeric types,
        ** including the types integer and Longint.
        -----------------------------------------------------------------}

        procedure I2X (x : integer;  var y : Extended);
        procedure S2X (x : Single;   var y : Extended);
        procedure D2X (x : Double;   var y : Extended);
        procedure C2X (x : Comp;     var y : Extended);
        procedure X2X (x : Extended; var y : Extended);
            { y := x (arithmetic assignment) }
```

```
        procedure X2I (x : Extended; var y : integer);
        procedure X2S (x : Extended; var y : Single);
        procedure X2D (x : Extended; var y : Double);
        procedure X2C (x : Extended; var y : Comp);
            { y := x (arithmetic assignment) }

{^ne 9 }
        {-----------------------------------------------------------
        ** These conversions apply to 68K systems only.  Longint is
        ** a 32-bit two's complement integer.
        -----------------------------------------------------------}

        procedure L2X (x : Longint;  var y : Extended);
        procedure X2L (x : Extended; var y : Longint);
            { y := x (arithmetic assignment) }

{^ne 17 }
        {-----------------------------------------------------------
        ** Conversions between the numeric types and the intermediate
        ** decimal type.
        -----------------------------------------------------------}

        procedure S2Dec (f : DecForm; x : Single;   var y : Decimal);
        procedure D2Dec (f : DecForm; x : Double;   var y : Decimal);
        procedure C2Dec (f : DecForm; x : Comp;     var y : Decimal);
        procedure X2Dec (f : DecForm; x : Extended; var y : Decimal);
            { y := x (according to the format f) }

        procedure Dec2S (x : Decimal; var y : Single);
        procedure Dec2D (x : Decimal; var y : Double);
        procedure Dec2C (x : Decimal; var y : Comp);
        procedure Dec2X (x : Decimal; var y : Extended);
            { y := x }

{^ne 18 }
        {-----------------------------------------------------------
        ** Conversions between the numeric types and strings.
        ** (These conversions have a built-in scanner/parser to convert
        ** between the intermediate decimal type and a string.)
        -----------------------------------------------------------}

        procedure S2Str (f : DecForm; x : Single;   var y : DecStr);
        procedure D2Str (f : DecForm; x : Double;   var y : DecStr);
        procedure C2Str (f : DecForm; x : Comp;     var y : DecStr);
        procedure X2Str (f : DecForm; x : Extended; var y : DecStr);
            { y := x (according to the format f) }

        procedure Str2S (x : DecStr; var y : Single);
        procedure Str2D (x : DecStr; var y : Double);
        procedure Str2C (x : DecStr; var y : Comp);
        procedure Str2X (x : DecStr; var y : Extended);
            { y := x }
```

```
{^ne 31 }
        {-----------  ------------------------------  -----------------
        ** Numerical 'library' procedures and functions.
        ------------------------------------------------------------}

        procedure RemX    (x : Extended; var y : Extended;
                                                  var quo : integer);
            { new y := remainder of ((old y) / x), such that
                       |new y| <= |x| / 2;
              quo := low order seven bits of integer quotient y / x,
                     so that -127 <= quo <= 127.                    }
        procedure SqrtX   (var x : Extended);
            { x := sqrt (x) }
        procedure RintX   (var x : Extended);
            { x := rounded value of x }
        procedure NegX    (var x : Extended);
            { x := -x }
        procedure AbsX    (var x : Extended);
            { x := |x| }
        procedure CpySgnX (var x : Extended; y : Extended);
            { x := x with the sign of y }

        procedure NextS   (var x : Single;   y : Single);
        procedure NextD   (var x : Double;   y : Double);
        procedure NextX   (var x : Extended; y : Extended);
            { x := next representable value from x toward y }

        function  ClassS  (x : Single;   var sgn : integer) : NumClass;
        function  ClassD  (x : Double;   var sgn : integer) : NumClass;
        function  ClassC  (x : Comp;     var sgn : integer) : NumClass;
        function  ClassX  (x : Extended; var sgn : integer) : NumClass;
            { sgn := sign of x (0 for pos, 1 for neg) }

        procedure ScalbX (n : integer; var y : Extended);
            { y := y * 2^n }
        procedure LogbX  (var x : Extended);
            { returns unbiased exponent of x }
{^ne 16 }
        {------------------------------------------------------------
        ** Manipulations of the static numeric state.
        ------------------------------------------------------------}

        procedure SetRnd  (r : RoundDir);
        procedure SetEnv  (e : Environ);
        procedure ProcExit(e : Environ);

        function  GetRnd  : RoundDir;
        procedure GetEnv  (var e : Environ);
        procedure ProcEntry (var e : Environ);

        function  TestXcp (x : Exception) : boolean;
        procedure SetXcp  (x : Exception; OnOff : boolean);
        function  TestHlt (x : Exception) : boolean;
        procedure SetHlt  (x : Exception; OnOff : boolean);
```

```
    {-------------------------------------------------------------------------}
{^sp 32767 }
    {-------------------------------------------------------------------------}

    IMPLEMENTATION

        {$I SANEIMP.TEXT}

END

{=========================================================================} .
```

```
{$C Copyright Apple Computer Inc., 1983 }

UNIT Elems;

    { Macintosh version. }

    {-----------------------------------------------------------------------}

    INTERFACE

        USES

            {$U OBJ:SANE.OBJ }

                SANE { Standard Apple Numeric Environment } ;

        procedure Log2X (var x : Extended);
            { x := log2 (x) }

        procedure LnX (var x : Extended);
            { x := ln (x) }

        procedure Ln1X (var x : Extended);
            { x := ln (1 + x) }

        procedure Exp2X (var x : Extended);
            { x := 2^x }

        procedure ExpX (var x : Extended);
            { x := e^x }

        procedure Exp1X (var x : Extended);
            { x := e^x - 1 }

        procedure XpwrI (i : integer; var x : Extended);
            { x := x^i }

        procedure XpwrY (y : Extended; var x : Extended);
            { x := x^y }

        procedure Compound (r, n : Extended; var x : Extended);
            { x := (1 + r)^n }

        procedure Annuity (r, n : Extended; var x : Extended);
            { x := (1 - (1 + r)^-n) / r }

        procedure SinX (var x : Extended);
            { x := sin(x) }

        procedure CosX (var x : Extended);
            { x := cos(x) }

        procedure TanX (var x : Extended);
            { x := tan(x) }
```

```
    procedure AtanX (var x : Extended);
        { x := atan(x) }

    procedure NextRandom (var x : Extended);
        { x := next random (x) }

{$p----------------------------------------------------------------------}
IMPLEMENTATION

    procedure Log2X { (var x : Extended) } ;                    EXTERNAL;
    procedure LnX { (var x : Extended) } ;                      EXTERNAL;
    procedure Ln1X { (var x : Extended) } ;                     EXTERNAL;
    procedure Exp2X { (var x : Extended) } ;                    EXTERNAL;
    procedure ExpX { (var x : Extended) } ;                     EXTERNAL;
    procedure Exp1X { (var x : Extended) } ;                    EXTERNAL;

    {
        Since Elems implementation expects pointer to integer argument,
        use this extra level of interface.
    }
    procedure XpwrIxxx(var i : integer; var x : Extended);  EXTERNAL;
    procedure XpwrI { (i : integer; var x : Extended) } ;
    begin
            XpwrIxxx(i, x);
    end;

    procedure XpwrY { (y : Extended; var x : Extended) } ;  EXTERNAL;
    procedure Compound { (r, n : Extended; var x : Extended) } ;  EXTERNAL;
    procedure Annuity { (r, n : Extended; var x : Extended) } ;  EXTERNAL;
    procedure SinX { (var x : Extended) } ;                     EXTERNAL;
    procedure CosX { (var x : Extended) } ;                     EXTERNAL;
    procedure TanX { (var x : Extended) } ;                     EXTERNAL;
    procedure AtanX { (var x : Extended) } ;                    EXTERNAL;
    procedure NextRandom { (var x : Extended) } ;               EXTERNAL;

END
{======================================================================}
{======================================================================}
{======================================================================}.
```

```
;-------------------------------------------------------
;
;  These macros give assembly language access to the Mac
;  floating-point arithmetic routines.  The arithmetic has
;  just one entry point.  It is typically accessed through
;  the tooltrap _FP68K, although a custom version of the
;  package may be linked as an object file, in which case
;  the entry point is the label %FP68K.
;
;  All calls to the arithmetic take the form:
;       PEA     <source address>
;       PEA     <destination address>
;       MOVE.W  <opcode>,-(SP)
;       _FP68K
;
;  All operands are passed by address.  The <opcode> word
;  specifies the instruction analogously to a 68000 machine
;  instruction.  Depending on the instruction, there may be
;  from one to three operand addresses passed.
;
;  This definition file specifies details of the <opcode>
;  word and the floating point state word, and defines
;  some handy macros.
;
;  Modification history:
;       29AUG82: WRITTEN BY JEROME COONEN
;       13OCT82: FB___ CONSTRANTS ADDED (JTC)
;       28DEC82: LOGB, SCALB ADDED, INF MODES OUT (JTC).
;       29APR83: ABS, NEG, CPYSGN, CLASS ADDED (JTC).
;       03MAY83: NEXT, SETXCP ADDED (JTC).
;       28MAY83: ELEMENTARY FUNCTIONS ADDED (JTC).
;       04JUL83: SHORT BRANCHES, TRIG AND RAND ADDED (JTC).
;       01NOV83: PRECISION CONTROL MADE A MODE (JTC).
;
;-------------------------------------------------------


;-------------------------------------------------------
; This constant determines whether the floating point unit
; is accessed via the system dispatcher after an A-line
; trap, or through a direct subroutine call to a  custom
; version of the package linked directly to the application.
;-------------------------------------------------------
ATRAP           .EQU    0       ;0 for JSR and 1 for A-line
BTRAP           .EQU    0       ;0 for JSR and 1 for A-line

        .MACRO  JSRFP
        .IF     ATRAP
        _FP68K
        .ELSE
        .REF    FP68K
        JSR     FP68K
        .ENDC
        .ENDM
```

```
        .MACRO  JSRELEMS
        .IF     BTRAP
        _ELEMS68K
        .ELSE
        .REF    ELEMS68K
        JSR     ELEMS68K
        .ENDC
        .ENDM


;---------------------------------------------------------
; OPERATION MASKS: bits $001F of the operation word
; determine the operation.  There are two rough classes of
; operations:  even numbered opcodes are the usual
; arithmetic operations and odd numbered opcodes are non-
; arithmetic or utility operations.
;---------------------------------------------------------
FOADD           .EQU    $0000
FOSUB           .EQU    $0002
FOMUL           .EQU    $0004
FODIV           .EQU    $0006
FOCMP           .EQU    $0008
FOCPX           .EQU    $000A
FOREM           .EQU    $000C
FOZ2X           .EQU    $000E
FOX2Z           .EQU    $0010
FOSQRT          .EQU    $0012
FORTI           .EQU    $0014
FOTTI           .EQU    $0016
FOSCALB         .EQU    $0018
FOLOGB          .EQU    $001A
FOCLASS         .EQU    $001C
; UNDEFINED     .EQU    $001E

FOSETENV        .EQU    $0001
FOGETENV        .EQU    $0003
FOSETTV         .EQU    $0005
FOGETTV         .EQU    $0007
FOD2B           .EQU    $0009
FOB2D           .EQU    $000B
FONEG           .EQU    $000D
FOABS           .EQU    $000F
FOCPYSGNX       .EQU    $0011
FONEXT          .EQU    $0013
FOSETXCP        .EQU    $0015
FOPROCENTRY     .EQU    $0017
FOPROCEXIT      .EQU    $0019
FOTESTXCP       .EQU    $001B
; UNDEFINED     .EQU    $001D
; UNDEFINED     .EQU    $001F


;---------------------------------------------------------
```

```
; OPERAND FORMAT MASKS: bits $3800 determine the format of
; any non-extended operand.
;-------------------------------------------------------------
FFEXT           .EQU    $0000   ; extended -- 80-bit float
FFDBL           .EQU    $0800   ; double   -- 64-bit float
FFSGL           .EQU    $1000   ; single   -- 32-bit float
FFINT           .EQU    $2000   ; integer  -- 16-bit integer
FFLNG           .EQU    $2800   ; long int -- 32-bit integer
FFCOMP          .EQU    $3000   ; accounting -- 64-bit int


;-------------------------------------------------------------
; Bit indexes for error and halt bits and rounding modes in
; the state word.  The word is broken down as:
;
;       $8000 -- unused
;
;       $6000 -- rounding modes
;                $0000 -- to nearest
;                $2000 -- toward +infinity
;                $4000 -- toward -infinity
;                $6000 -- toward zero
;
;       $1F00 -- error flags
;                $1000 -- inexact
;                $0800 -- division by zero
;                $0400 -- overflow
;                $0200 -- underflow
;                $0100 -- invalid operation
;
;       $0080 -- result of last rounding
;                $0000 -- rounded down in magnitude
;                $0080 -- rounded up in magnitude
;
;       $0060 -- precision control
;                $0000 -- extended
;                $0020 -- double
;                $0040 -- single
;                $0060 -- ILLEGAL
;
;       $001F -- halt enables, corresponding to error flags
;
; The bit indexes are based on the byte halves of the state
; word.
;-------------------------------------------------------------
FBINVALID       .EQU    0       ; invalid operation
FBUFLOW         .EQU    1       ; underflow
FBOFLOW         .EQU    2       ; overflow
FBDIVZER        .EQU    3       ; division by zero
FBINEXACT       .EQU    4       ; inexact
FBRNDLO         .EQU    5       ; low bit of rounding mode
FBRNDHI         .EQU    6       ; high bit of rounding mode
FBLSTRND        .EQU    7       ; last round result bit
FBDBL           .EQU    5       ; double precision control
```

```
FBSGL           .EQU    6       ; single precision control

;---------------------------------------------------------
; FLOATING CONDITIONAL BRANCHES: floating point comparisons
; set the CPU condition code register (the CCR) as follows:
;       relation        X N Z V C
;       ------------------------
;         equal         0 0 1 0 0
;       less than       1 1 0 0 1
;       greater than    0 0 0 0 0
;        unordered      0 0 0 1 0
; The macros below define a set of so-called floating
; branches to spare the programmer repeated refernces to the
; the table above.
;---------------------------------------------------------
        .MACRO  FBEQ
        BEQ     %1
        .ENDM

        .MACRO  FBLT
        BCS     %1
        .ENDM

        .MACRO  FBLE
        BLS     %1
        .ENDM

        .MACRO  FBGT
        BGT     %1
        .ENDM

        .MACRO  FBGE
        BGE     %1
        .ENDM

        .MACRO  FBULT
        BLT     %1
        .ENDM

        .MACRO  FBULE
        BLE     %1
        .ENDM

        .MACRO  FBUGT
        BHI     %1
        .ENDM

        .MACRO  FBUGE
        BCC     %1
        .ENDM

        .MACRO  FBU
        BVS     %1
```

```
        .ENDM

        .MACRO  FBO
        BVC     %1
        .ENDM

        .MACRO  FBNE
        BNE     %1
        .ENDM

        .MACRO  FBUE
        BEQ     %1
        BVS     %1
        .ENDM

        .MACRO  FBLG
        BNE     %1
        BVC     %1
        .ENDM

; Short branch versions.

        .MACRO  FBEQS
        BEQ.S   %1
        .ENDM

        .MACRO  FBLTS
        BCS.S     %1
        .ENDM

        .MACRO  FBLES
        BLS.S     %1
        .ENDM

        .MACRO  FBGTS
        BGT.S     %1
        .ENDM

        .MACRO  FBGES
        BGE.S     %1
        .ENDM

        .MACRO  FBULTS
        BLT.S     %1
        .ENDM

        .MACRO  FBULES
        BLE.S     %1
        .ENDM

        .MACRO  FBUGTS
        BHI.S     %1
        .ENDM
```

```
        .MACRO  FBUGES
        BCC.S     %1
        .ENDM

        .MACRO  FBUS
        BVS.S     %1
        .ENDM

        .MACRO  FBOS
        BVC.S     %1
        .ENDM

        .MACRO  FBNES
        BNE.S     %1
        .ENDM

        .MACRO  FBUES
        BEQ.S     %1
        BVS.S     %1
        .ENDM

        .MACRO  FBLGS
        BNE.S     %1
        BVC.S     %1
        .ENDM


;-----------------------------------------------------------
; OPERATION MACROS:
;       THESE MACROS REQUIRE THAT THE OPERANDS' ADDRESSES
;       FIRST BE PUSHED ON THE STACK.  THE MACROS CANNOT
;       THEMSELVES PUSH THE ADDRESSES SINCE THE ADDRESSES
;       MAY BE SP-RELATIVE, IN WHICH CASE THEY REQUIRE
;       PROGRAMMER CARE.
; OPERATION MACROS: operand addresses should already be on
; the stack, with the destination address on top.  The
; suffix X, D, S, or C determines the format of the source
; operand -- extended, double, single, or computational
; respectively; the destination operand is always extended.
;-----------------------------------------------------------


;-----------------------------------------------------------
; Addition.
;-----------------------------------------------------------
        .MACRO  FADDX
        MOVE.W  #FFEXT+FOADD,-(SP)
        JSRFP
        .ENDM

        .MACRO  FADDD
        MOVE.W  #FFDBL+FOADD,-(SP)
        JSRFP
        .ENDM
```

```
        .MACRO  FADDS
        MOVE.W  #FFSGL+FOADD,-(SP)
        JSRFP
        .ENDM

        .MACRO  FADDC
        MOVE.W  #FFCOMP+FOADD,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------------
; Subtraction.
;-------------------------------------------------------------
        .MACRO  FSUBX
        MOVE.W  #FFEXT+FOSUB,-(SP)
        JSRFP
        .ENDM

        .MACRO  FSUBD
        MOVE.W  #FFDBL+FOSUB,-(SP)
        JSRFP
        .ENDM

        .MACRO  FSUBS
        MOVE.W  #FFSGL+FOSUB,-(SP)
        JSRFP
        .ENDM

        .MACRO  FSUBC
        MOVE.W  #FFCOMP+FOSUB,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------------
; Multiplication.
;-------------------------------------------------------------
        .MACRO  FMULX
        MOVE.W  #FFEXT+FOMUL,-(SP)
        JSRFP
        .ENDM

        .MACRO  FMULD
        MOVE.W  #FFDBL+FOMUL,-(SP)
        JSRFP
        .ENDM

        .MACRO  FMULS
        MOVE.W  #FFSGL+FOMUL,-(SP)
        JSRFP
        .ENDM
```

```
        .MACRO  FMULC
        MOVE.W  #FFCOMP+FOMUL,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------
; Division.
;-------------------------------------------------------
        .MACRO  FDIVX
        MOVE.W  #FFEXT+FODIV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDIVD
        MOVE.W  #FFDBL+FODIV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDIVS
        MOVE.W  #FFSGL+FODIV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDIVC
        MOVE.W  #FFCOMP+FODIV,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------
; Compare, signaling no exceptions.
;-------------------------------------------------------
        .MACRO  FCMPX
        MOVE.W  #FFEXT+FOCMP,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCMPD
        MOVE.W  #FFDBL+FOCMP,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCMPS
        MOVE.W  #FFSGL+FOCMP,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCMPC
        MOVE.W  #FFCOMP+FOCMP,-(SP)
        JSRFP
        .ENDM
```

```
;-------------------------------------------------------
; Compare, signaling invalid operation if the two operands
; are unordered.
;-------------------------------------------------------
        .MACRO  FCPXX
        MOVE.W  #FFEXT+FOCPX,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCPXD
        MOVE.W  #FFDBL+FOCPX,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCPXS
        MOVE.W  #FFSGL+FOCPX,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCPXC
        MOVE.W  #FFCOMP+FOCPX,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------
; Remainder.  The remainder is placed in the destination,
; and the low bits of the integer quotient are placed in
; the low word of register D0.
;-------------------------------------------------------
        .MACRO  FREMX
        MOVE.W  #FFEXT+FOREM,-(SP)
        JSRFP
        .ENDM

        .MACRO  FREMD
        MOVE.W  #FFDBL+FOREM,-(SP)
        JSRFP
        .ENDM

        .MACRO  FREMS
        MOVE.W  #FFSGL+FOREM,-(SP)
        JSRFP
        .ENDM

        .MACRO  FREMC
        MOVE.W  #FFCOMP+FOREM,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------
; Compare the source operand to the extended format and
; place in the destination.
```

```
;-------------------------------------------------------------
        .MACRO  FX2X
        MOVE.W  #FFEXT+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO  FD2X
        MOVE.W  #FFDBL+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO  FS2X
        MOVE.W  #FFSGL+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO  FI2X                    ; 16-bit integer
        MOVE.W  #FFINT+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO  FL2X                    ; 32-bit integer
        MOVE.W  #FFLNG+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO  FC2X
        MOVE.W  #FFCOMP+FOZ2X,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------------
; Convert the extended source operand to the specified
; format and place in the destination.
;-------------------------------------------------------------
        .MACRO  FX2D
        MOVE.W  #FFDBL+FOX2Z,-(SP)
        JSRFP
        .ENDM

        .MACRO  FX2S
        MOVE.W  #FFSGL+FOX2Z,-(SP)
        JSRFP
        .ENDM

        .MACRO  FX2I                    ; 16-bit integer
        MOVE.W  #FFINT+FOX2Z,-(SP)
        JSRFP
        .ENDM

        .MACRO  FX2L                    ; 32-bit integer
        MOVE.W  #FFLNG+FOX2Z,-(SP)
        JSRFP
```

```
        .ENDM

        .MACRO  FX2C
        MOVE.W  #FFCOMP+FOX2Z,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------------
; Miscellaneous operations applying only to extended
; operands.  The input operand is overwritten with the
; computed result.
;-------------------------------------------------------------

; Square root.
        .MACRO  FSQRTX
        MOVE.W  #FOSQRT,-(SP)
        JSRFP
        .ENDM

; Round to integer, according to the current rounding mode.
        .MACRO  FRINTX
        MOVE.W  #FORTI,-(SP)
        JSRFP
        .ENDM

; Round to integer, forcing rounding toward zero.
        .MACRO  FTINTX
        MOVE.W  #FOTTI,-(SP)
        JSRFP
        .ENDM

; Set the destination to the product:
;  (destination) * 2^(source)
; where the source operand is a 16-bit integer.
        .MACRO  FSCALBX
        MOVE.W  #FFINT+FOSCALB,-(SP)
        JSRFP
        .ENDM

; Replace the destination with its exponent, converted to
; the extended format.
        .MACRO  FLOGBX
        MOVE.W  #FOLOGB,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------------
; Non-arithmetic sign operations on extended operands.
;-------------------------------------------------------------

; Negate.
        .MACRO  FNEGX
```

```
        MOVE.W  #FONEG,-(SP)
        JSRFP
        .ENDM

; Absolute value.
        .MACRO  FABSX
        MOVE.W  #FOABS,-(SP)
        JSRFP
        .ENDM

; Copy the sign of the destination operand onto the sign of
; the source operand.  Note that the source operand is
; modified.
        .MACRO  FCPYSGNX
        MOVE.W  #FOCPYSGN,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------
; The nextafter operation replaces the source operand with
; its nearest representable neighbor in the direction of the
; destination operand.  Note that both operands are of the
; the same format, as specified by the usual suffix.
;-------------------------------------------------------
        .MACRO  FNEXTS
        MOVE.W  #FFSGL+FONEXT,-(SP)
        JSRFP
        .ENDM

        .MACRO  FNEXTD
        MOVE.W  #FFDBL+FONEXT,-(SP)
        JSRFP
        .ENDM

        .MACRO  FNEXTX
        MOVE.W  #FFEXT+FONEXT,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------
; The classify operation places an integer in the
; destination.  The sign of the integer is the sign of the
; source.  The magnitude is determined by the value of the
; source, as indicated by the equates.
;-------------------------------------------------------
FCSNAN          .EQU    1       ; signaling NAN
FCQNAN          .EQU    2       ; quiet NAN
FCINF           .EQU    3       ; infinity
FCZERO          .EQU    4       ; zero
FCNORM          .EQU    5       ; normal number
FCDENORM        .EQU    6       ; denormal number
```

```
        .MACRO  FCLASSS
        MOVE.W  #FFSGL+FOCLASS,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCLASSD
        MOVE.W  #FFDBL+FOCLASS,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCLASSX
        MOVE.W  #FFEXT+FOCLASS,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCLASSC
        MOVE.W  #FFCOMP+FOCLASS,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------
; These four operations give access to the floating point
; state (or environment) word and the halt vector address.
; The sole input operand is a pointer to the word or address
; to be placed into the arithmetic state area or read from
; it.
;-------------------------------------------------------
        .MACRO  FGETENV
        MOVE.W  #FOGETENV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FSETENV
        MOVE.W  #FOSETENV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FGETTV
        MOVE.W  #FOGETTV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FSETTV
        MOVE.W  #FOSETTV,-(SP)
        JSRFP
        .ENDM

;-------------------------------------------------------
; Both FPROCENTRY and FPROCEXIT have one operand -- a
; pointer to a word.  The entry procedure saves the current
; floating point state in that word and resets the state
; to 0, that is all modes to default, flags and halts to
; OFF.  The exit procedure performs the sequence:
```

```
;       1. Save current error flags in a temporary.
;       2. Restore the state saved at the address given by
;               the parameter.
;       3. Signal the exceptions flagged in the temporary,
;               halting if so specified by the newly
;               restored state word.
; These routines serve to handle the state word dynamically
; across subroutine calls.
;-------------------------------------------------------------
        .MACRO  FPROCENTRY
        MOVE.W  #FOPROCENTRY,-(SP)
        JSRFP
        .ENDM

        .MACRO  FPROCEXIT
        MOVE.W  #FOPROCEXIT,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------------
; FSETXCP is a null arithmetic operation which stimulates
; the indicated exception.  It may be used by library
; routines intended to behave like elementary operations.
; The operand is a pointer to an integer taking any value
; between FBINVALID and FBINEXACT.
; FTESTXCP tests the flag indicated by the integer pointed
; to by the input address.  The integer is replaced by a
; Pascal boolean (word $0000=false, $0100=true)
;-------------------------------------------------------------
        .MACRO  FSETXCP
        MOVE.W  #FOSETXCP,-(SP)
        JSRFP
        .ENDM

        .MACRO  FTESTXCP
        MOVE.W  #FOTESTXCP,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------------
; WARNING: PASCAL ENUMERATED TYPES, LIKE THOSE OF THE
; DECIMAL RECORD, ARE STORED IN THE HIGH-ORDER BYTE OF THE
; ALLOCATED WORD, IF POSSIBLE.  THUS THE SIGN HAS THE
; INTEGER VALUE 0 FOR PLUS AND 256 (RATHER THAN 1)
; FOR MINUS.
; BINARY-DECIMAL CONVERSION:  The next routines convert
; between a canonical decimal format and the binary format
; specified.  The decimal format is defined in Pascal as
;
;   CONST
;       SIGDIGLEN = 20;
;
```

```
;   TYPE
;       SigDig  = string [SIGDIGLEN];
;       Decimal = record
;                     sgn : 0..1;
;                     exp : integer;
;                     sig : SigDig
;                 end;
;
; Note that Lisa Pascal stores the sgn in the high-order
; byte of the allotted word, so the two legal word values
; of sgn are 0 and 256.
;-----------------------------------------------------------


;-----------------------------------------------------------
; Decimal to binary conversion is governed by a format
; record defined in Pascal as:
;
;   TYPE
;       DecForm = record
;                     style  : (FloatDecimal, FixedDecimal);
;                     digits : integer
;                 end;
;
; Note again that the style field is stored in the high-
; order byte of the allotted word.
;
; These are the only operations with three operands.  The
; pointer to the format record is deepest in the stack,
; then the source pointer, and finally the destination
; pointer.
;-----------------------------------------------------------
        .MACRO  FDEC2X
        MOVE.W  #FFEXT+FOD2B,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDEC2D
        MOVE.W  #FFDBL+FOD2B,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDEC2S
        MOVE.W  #FFSGL+FOD2B,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDEC2C
        MOVE.W  #FFCOMP+FOD2B,-(SP)
        JSRFP
        .ENDM


;-----------------------------------------------------------
```

```
; Binary to decimal conversion.
;-------------------------------------------------------
        .MACRO  FX2DEC
        MOVE.W  #FFEXT+FOB2D,-(SP)
        JSRFP
        .ENDM

        .MACRO  FD2DEC
        MOVE.W  #FFDBL+FOB2D,-(SP)
        JSRFP
        .ENDM

        .MACRO  FS2DEC
        MOVE.W  #FFSGL+FOB2D,-(SP)
        JSRFP
        .ENDM

        .MACRO  FC2DEC
        MOVE.W  #FFCOMP+FOB2D,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------
; Equates and macros for elementary functions.
;-------------------------------------------------------
FOLNX           .EQU    $0000
FOLOG2X         .EQU    $0002
FOLN1X          .EQU    $0004
FOLOG21X        .EQU    $0006

FOEXPX          .EQU    $0008
FOEXP2X         .EQU    $000A
FOEXP1X         .EQU    $000C
FOEXP21X        .EQU    $000E

FOXPWRI         .EQU    $8010
FOXPWRY         .EQU    $8012
FOCOMPOUNDX     .EQU    $C014
FOANNUITYX      .EQU    $C016

FOSINX          .EQU    $0018
FOCOSX          .EQU    $001A
FOTANX          .EQU    $001C
FOATANX         .EQU    $001E
FORANDOMX       .EQU    $0020

        .MACRO  FLNX
        MOVE.W  #FOLNX,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FLOG2X
        MOVE.W  #FOLOG2X,-(SP)
```

```
JSRELEMS
.ENDM

.MACRO  FLN1X
MOVE.W  #FOLN1X,-(SP)
JSRELEMS
.ENDM

.MACRO  FLOG21X
MOVE.W  #FOLOG21X,-(SP)
JSRELEMS
.ENDM

.MACRO  FEXPX
MOVE.W  #FOEXPX,-(SP)
JSRELEMS
.ENDM

.MACRO  FEXP2X
MOVE.W  #FOEXP2X,-(SP)
JSRELEMS
.ENDM

.MACRO  FEXP1X
MOVE.W  #FOEXP1X,-(SP)
JSRELEMS
.ENDM

.MACRO  FEXP21X
MOVE.W  #FOEXP21X,-(SP)
JSRELEMS
.ENDM

.MACRO  FXPWRI
MOVE.W  #FOXPWRI,-(SP)
JSRELEMS
.ENDM

.MACRO  FXPWRY
MOVE.W  #FOXPWRY,-(SP)
JSRELEMS
.ENDM

.MACRO  FCOMPOUNDX
MOVE.W  #FOCOMPOUNDX,-(SP)
JSRELEMS
.ENDM

.MACRO  FANNUITYX
MOVE.W  #FOANNUITYX,-(SP)
JSRELEMS
.ENDM

.MACRO  FSINX
```

```
        MOVE.W  #FOSINX,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FCOSX
        MOVE.W  #FOCOSX,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FTANX
        MOVE.W  #FOTANX,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FATANX
        MOVE.W  #FOATANX,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FRANDOMX
        MOVE.W  #FORANDOMX,-(SP)
        JSRELEMS
        .ENDM
;-------------------------------------------------------------
;-------------------------------------------------------------
;-------------------------------------------------------------
;-------------------------------------------------------------
```

Introduction

FP68K provides conversions between the extended floating-point format and three integer formats:

```
int16  -- 16-bit two's complement
int32  -- 32-bit two's complement
comp64 -- 64-bit two's complement with the reserved value
          hexadecimal 8000000000000000.
```

One Pascal program, ITBATTERY.TEXT, tests all three conversions. This document describes how to use and, if necessary, modify the tests.

Compiling and running

ITBATTERY.TEXT uses the SANE interface (see the "High Level Interface" document, so it must be linked with the SANE object files, as well as with the usual nonarithmetic Pascal run-time libraries (e.g. *MPASLIB on Lisa). The program will simply run to completion, with a Pascal HALT if an error is found; execution time may run to 15 minutes on a Lisa system.

What is tested

Each of the integer formats is tested in two phases. First, a collection of specific extended numbers is converted to the integer format, with tests for correct rounding and signaling of the invalid exception when appropriate. Then a set of

```
integer --> extended --> integer
```

conversions is run, with the input and output integers compared for equality. In the case of int16, all 2^16 cases are run. However exhaustive testing of int32 and comp64 is infeasible so a loop is set up to do 2^16 tests from several starting points.

Introduction

The most important and rigorous set of tests of FP68K is the set of so-called IEEE test vectors. These tests, developed by the author while at Zilog, are used to test implementations of proposed standard P754. They were donated to the IEEE subcommittee 754 by Zilog Inc., and are now distributed by that subcommittee. The tests have undergone major revision within Apple, thanks especially to Jim Thomas of PCS.

Form of the tests

Each vector is an ascii string describing an operation, operands, and the result. For example, "lincl" is the floating-point number (of the format under consideration) next larger than 1. When "1" is subtracted from "lincl", the result is "lulpl", just one unit in the last place of 1. Written this way, the vectors may be applied to any floating-point format. The tests carefully inspect the nuances of rounding and exception handling. A document is under development to explain in detail the next release of the test vectors, scheduled for early 1983, after some last details of the standard are cleared up.

Files

The test vectors are contained in a family of files by the name of TVxxxx.2.TEXT and TWxxx.2.TEXT. The "2" refers to version 2 of the tests. (Version 1 was based on Draft 8.0 of the standard.) The file TLIST.TEXT is a list of the test file names to be used in any given run of the test. Pascal file TD68.TEXT with unit TD68FP.TEXT actually run the tests. These interface with FP68K exclusively through the SANE interface.

Introduction

Since the binary <--> conversions within FP68K approximate the mathematical identity operation, they lend themselves to certain types of self-testing. For example, if enough decimal digits are kept, then the conversion

binary --> decimal --> binary

is the identity mapping when results are rounded to nearest. The number of digits required turns out to be 9 for single and 17 for double. A similar test performs the first conversion rounding toward plus infinity and the second rounding toward minus infinity. In this case the final result may differ from the starting value by one unit in the direction of the latter rounding, so the program allows this discrepancy.

This document describes the test files and how they can be run. For details of the underlying error analysis (which is quite subtle) see the paper "Accurate Yet Economical Binary-Decimal Conversions" by J. Coonen.

Test programs

The test programs are:

IOS.TEXT
IOSF.TEXT
IOD.TEXT
IODF.TEXT
IONAN.TEXT
IOPSCAN.TEXT

The letter "S" and "D" distinguishes single and double tests. The IOS.TEXT and IOD.TEXT tests run with both rounding to nearest and the directed roundings. The "F" tests use fixed-format output rather than floating-format output for the intermediate decimal string. The IOPSCAN test is used to check the preformance of the printer and scanner used by SANE68, and included from file SAPSCAN.TEXT. The IONAN test checks the input and output conversion of some 20 stock NANs, and then allows the user to enter any decimal string to be converted to the three formats in three rounding modes. Neither IOPSCAN nor IONAN are self-checking; rather, the user must monitor their output.

The tests cover extreme intervals where the decimal numbers are sparsest and densest with respect to the binary numbers. Sparse intervals have the form [10^N, 2^n] where the endpoints are nearly equal. Dense intervals have the corresponding form [2^m, 10^M].

Running the tests

Each of the programs is compiled and run separately. The programs use the SANE interface. A test will HALT with a suitable diagnostic if the test

fails.

The single format cases are few enough that their tests can be run overnight. However, the double format cases will run essentially forever since the number of interesting cases is so great. A few overnight tests should be sufficient.

Background

The so-called I/O routines for scanning and printing floating-point numbers in decimal form are complicated by subtle numerical issues and nettlesome design decisions. For example, even the simplest, stripped-down conversion routines require over one-third the code space (about 1.3K) of the rest of the FP68K binary floating-point package. With a full parser and formatter, the conversion routines are much larger. And it is unclear whether full routines would be flexible enough for use in different language systems and I/O-intensive applications like Visi-Calc.

Where does the responsibility lie? This note argues that the core conversion routines, which are part of the arithmetic package, should be kept very simple. Above them -- somewhere in the system -- should be a full scanner and formatter available to languages and applications, but not forced upon them. This would lead to the most efficient use of code space and execution time.

The Sad Truth

Numerical I/O can be monstrous. Since each computer language has its own grammar for floating-point numbers and its own conventions for output format, it almost necessary for each language system on a computer to provide significant I/O support. Unfortunately, this may be layered upon the host system's I/O system. And it is not unusual (Apple III, for example) for a language compiler to use different conversion routines than the I/O system the compiled code utilizes.

In another case, designers of the UNIX operating system attempted to route all conversion through the routines atof(), ascii to floating, and the pair ecvt(), fcvt() for floating and fixed conversion to ascii. But even this fairly clean design has led to VERY complicated software shells around atof, ecvt, and fcvt. Numerical accuracy aside, the complexity of just the character hacking is forbidding.

One problem with the UNIX design lies in its failure to properly divide responsibility for the distinct processes involved in conversion, namely:

1. Recognize floating-point strings (in compilers, ...)
2. Translate strings to numerical values.
3. Determine which output format (fixed or floating) is appropriate for a given value.
4. Translate a numerical value to a string.

The utilities atof() and ecvt() provide items 2 and 4. Item 3, printing a number in its "nicest" form is provided in rough form through ecvt(). But recognizing strings is left to each language compiler's lexical scanner. Unfortunately, after a scanner has parsed a floating decimal string, it passes it along to atof() where it is parsed once more.

A Proposal for Change

(1) Support, at the arithmetic level, conversions between each of the available binary floating-point types and one decimal structure describable in Pascal as:

```
{*
** Low-level format of the floating decimal value:
**   (-1)^sgn * 10^exp * dig
** The constant DECSTRLEN is 20 for MAC and 28 for III, since
** the latter uses very high precision for intermediates.
*}
type
    DecStr  = string[DECSTRLEN]
    Decimal = record
        sgn: 0..1;      {0 for +, 1 for -}
        exp: integer;
        sig: DecStr
    end;
```

(2) Rigidly specify the format of Decimal.sig for decimal to binary conversions, relying upon a lexical scanner to perform the first parse. The decimal value would depend upon the first character of decrec.dig:

```
'I'        --> infinity
'Nxxx...x' --> NAN, with optional ascii hex digits 0-9, A-F, a-f
'0'        --> zero
'ddd...d'  --> string of digits stripped of leading and trailing zeros
```

The digit string would never be more than 20 digits long. If present, the 20-th digit would indicate the absence of nonzero trailing digits beyond the 20-th (to aid in correct rounding).

(3) Specify decimal output format through a structure like the Pascal:

```
{*
** Output format specifier.
*}
type
    DecForm = record
        style: (float, fixed);
        digits: integer
    end;
```

For "float" conversions, digits is the number of significant digits to be delivered in Decimal.sig. For "fixed" conversions, count is the number of fraction digits to be converted (a negative count suppresses conversion of low-order integer digits).

Sometimes it is desired to print a number in the nicest form possible for a given field width. For example, the string "1.23456789" conveys much more information in 10 characters than does "1.2345e+04". Such conversions are

discussed in the next section.

(4) Provide a scanner and formatter which, if not of most general use, provide models that can be tailored to a particular application. Samples are built into the implementation section of the SANE Pascal interface; they are contained in the file SAPSCAN.TEXT.

Binary --> Decimal

The family of routines:

S2Dec
D2Dec
X2Dec
C2Dec

provide conversions to the Decimal record format described above. Special cases are keyed by the first character of Decimal.sig:

'0' : zero
'I' : infinity
'N' : not-a-number, followed by optional ascii hex digits; if there are fewer than four, they are padded on the left with 0's.
'?' : overflow of fixed-style format

These must be used with a formatter to produce output strings.

The family:

S2Str
D2Str
X2Str
C2Str

uses the built-in formatter, Dec2Str, to generate ascii string output.

Decimal --> Binary

These conversions are povided by the complementary set of procedures: Dec2S, Dec2D, Dec2X, Dec2C, and Str2S, Str2D, Str2X, Str2C. In the case of the Dec2* conversions, the first character of Decimal.sig indicates special cases as noted above for *2Dec conversions.

Infinity and NAN conversions

Infinity is printed and read as a string of sign characters, "+++++" or "-----".

On input, NANs have the general form NAN'xxxx:yyy...y'. The x's and y's should be ascii hex digits: 0-9, A-F, a-f. The string portion following NAN may be omitted. The x's are padded on the LEFT with 0's to width 4. The y's are padded on the RIGHT with 0's to the width of the NAN's significant bit

field.

On output, NANs will be printed in the same format. Leading x=0 and trailing y=0 are omitted, but at least one x is printed. If all y=0, then the colon and the y field is dropped.

Any unrecognizable string is converted to a NAN.

Background

Applications like accounting spreadsheets typically need to display floating-point values in decimal form within a field of fixed width. For maximum readability, the output should be in integer or fixed-point format if possible, with floating-point format as a last resort. The idea is to avoid listing small integers in the abominable form 0.100000000000E1 reminiscent of computing in the McCarthy era.

The problem

Given a binary floating-point number X and an ascii field F, display X in the "nicest", most informative way within F.

A proposal

1. If X may be displayed in a subfield of F, pad X on the left with blanks.

2. Display the sign of X only if it is '-'.

3. If X is an integer and F is wide enough to accommodate X, then display X as an integer, without a trailing '.'.

4. Else if X has nonzero integer and fraction parts and F is wide enough to accommodate at least the integer part of F and its trailing '.', then display X in the fixed-point form ZZZZ.YYYY with as many fraction digits as F will accommodate, up to a maximum of 17 significant digits.

5. Else if $|X| < 1$ and F is wide enough that X may be displayed in the form 0.00000ZZZZZ with no more 0s just to the right of the decimal point than digits following those 0s, then display X in that fixed-point form with up to 17 significant digits.

6. Finally, if all the above fail, then display X in the floating-point form Z.ZZZZZEYYY with as many significant digits up to 17 as F will accommodate, taking into account the width of the exponent field, including its possible sign. Display the sign of the exponent field only if it is '-'.

An implementation

The above choices depend on detailed knowledge of the magnitude of X. For example, in producing floating-point output, it is necessary to know the number of spaces that will be occupied by the decimal exponent (with sign, it could be 1 to 5) in order to know how many significant digits to which to round X. In the worst case, this could mean several calls upon the low-level conversion routine until the proper output is finally obtained.

One easy way to bypass these problems, and keep the fundamental conversion routine simple, is perform the binary -> decimal conversion in two stages. First convert the binary value X to the SANE decimal form:

```
type
    DecStr  = string[DECSTRLEN];    { length is 20 for MAC }
    Decimal = record
        sgn: integer;    {0 for +, nonzero for -}
        exp: integer;    {as though decimal is at the right of...}
        sig: DecStr
    end;
```

If the conversion is performed with rounding toward 0, conversion style = float, and digit count = 19, and if the inexact exception flag is cleared before the conversion, then the 19-digit result may be correctly rounded to the desired width after the ultimate output format is determined. Since no more than 17 digits will ever be displayed (recall that 17 digits suffice to distinguish double format binary numbers), the 19 digits together with the inexact exception flag permit correct rounding.

The second step of the conversion decides, on the basis of the intermediate decimal form, which format is appropriate. Then the decimal value is rounded (in decimal!) and displayed as desired. Note that this scheme has as a happy byproduct the ability to round in the (time-honored?) "add half and chop" manner that is unavailable within Apple arithmetic itself.

In the interest of compatibility of the floating-point arithmetic on Apples II/III and Mac/(Lisa?), the following GRITTY DETAILS were discussed on June 29. This is an update on the decisions made then.

1. Distinguishing signaling and quiet NANs: use the leading fraction bit, 0-quiet and 1-signaling.

2. Explicit leading bit of extended NANs and INFs: ignore it, that is decide whether NAN or INF on the basis of the fraction bits only.

3. Quiet NANs have an 8-bit "indicator field" marked by stars in the following extended format hex mask: XXXX XX** XXXX XXXX XXXX. This byte is the low half of the leading word of significant bits. The interpretation of the field is as given page 70 of Apple III Pascal, volume 2, subject to enhancements.

4. When two quiet NANs are operands to the operations +, -, *, /, and REM, one or the other of the NANs is output. When the indicator fields differ, the NAN with the larger indicator field prevails; ties are broken arbitrarily.

5. True to the standard, the sign of an output NAN is unspecified.

6. Signaling NANs precipitate the invalid operation exception when they appear as operands.

7. Underflow is tested before rounding. CHANGE: this may change depending on P754 deliberations in the late summer of '82

8. Projective INF follows the same rules of signs as affine INF. The ABSOLUTELY ONLY differences between affine and projective modes are: the UNORDERED-ness of projective INF in comparisons with finite numbers, and the invalid operation exception that arises from the sum of two projective INFs with the same sign. CHANGE: projective mode may be removed from P754 in late summer '82.

9. Treatment of unnormalized extended numbers may differ between systems. 68K implementations will normalize all such, as is expected of the Motorola and Zilog chips. 6502 implementations may support the ANTIQUE warning mode in preliminary releases, though it may never be documented for general consumption.

10. The bottom of the extended exponent range is as in the Motorola and Zilog implementations (as opposed to Intel). That is, there is no redundancy between the bottom two exponent values.

11. The exponent bias in extended is hex 3FFF, which is used by Intel, Zilog, and Motorola. Motorola may insert a word of garbage between the sign/exp fields and the significant bits in order to have a 96-bit data type.

12. Comparisons return results according to local system convenience. 68K: return from the floating-point software with the CPU condition codes set appropriately for a conditional branch. 6502: for lack of a rich set of conditional branches, let the comparison operation be a family of boolean tests like "Is X <= Y?" The difference between the two systems should be

hidden well below the high-level language interfaces.

13. Auxiliary functions: relegate functions like nextafter() to the system numerical library rather than putting them in the arithmetic engine.

14. The data types specified by SANE are int16, comp32, comp64, f32, f64, x80. 68K systems will require int32 as well.

15. Is the Pascal assignment: X := Y; an arithmetic operation when both X and Y are variables of the same floating-point format? Or is a straight byte copy sufficient? This is really a language issue -- one left dangling by the standard. The arithmetic units, if asked to perform a floating move between two floating entities of the same format, will perform a full-blown arithmetic operation. This will cause side effects if the floating value is a signaling NAN (invalid operation) or a denormalized number (underflow).

16. Precision control is supported by 6502 and 68K packages, but it is available only through assembly language -- it is intended only for SPECIAL applications anyway. Precision control implies range control, too.

17. There is no "integer overflow" exception.

18. Traps? These are so system-dependent there is no hope for perfect consistency. So the issue is left as a local matter for each system. The question relevant to each floating-point engine is: "What information will I be required to spew out in case of a trap?"

TABLE OF CONTENTS

1. Design Philosophy

The software package FP68K provides binary floating-point arithmetic according to the proposed IEEE standard P754. This arithmetic is in turn the basis for SANE, standard Apple numeric environment. The goal is software compatibility between the various Apple products supporting SANE.

The arithmetic package is reasonably small and fast. Its interface is very simple. And it provides just those operations needed for applications software. Although developed specifically for Mac, the package is designed for use in Lisa, if desired.

The following sequence of examples illustrates the SANE philosophy:

Single operation: X + Y

P754: Compute as if with unbounded range and precision, then coerce to destination format.

Expression evaluation: Z := (X + Y)/(U + V);

SANE: Compute all anonymous intermediate subexpressions to extended, then coerce to destination format.

Loop:

```
S := 0.0;
FOR I := 1 TO N DO
    S := S + A[I]*B[I];
```

VERY SANE: Wise programmer uses extended variable S to eliminate spurious over/underflows in the inner loop, and to reduce the final rounding error.

2. Data Types

The arithmetic supports the following data types. All are specified in SANE except for int32 and decimal. Int32 is included for convenience in 68K environments, where 32-bit integers are common. Through the decimal type the package provides the basis for the binary<->decimal conversions required by languages and the I/O system.

```
int16    --  16-bit two's-complement integer
int32    --  32-bit two's-complement integer
comp64   --  64-bit integer, with one reserved operand value
f32      --  32-bit single floating-point
f64      --  64-bit double floating-point
x80      --  80-bit extended floating-point
decimal  --  ascii digit string with integer sign and exponent
```

3. Arithmetic Operations

These operations apply to floating-point operands:

```
+, -, *, /, SQRT, REMAINDER, COMPARE,
ROUND TO INTEGER, TRUNCATE TO INTEGER, LOGB, SCALB,
ABSOLUTE VALUE, NEGATE, COPYSIGN, NEXAFTER, CLASS
```

Except for COMPARE, each produces a floating-point result. COMPARE sets the CPU flag bits according to the two operands. Besides its floating-point result, REMAINDER returns the sign and four least significant bits of its integer quotient in the CPU flags (a very useful trick for argument reduction in the transcendental functions). LOGB replaces a number by is unbiased exponent, in floating form; SCALB scales a number by an integer power of 2.

4. Format Conversions

```
intXX     <-->  extended
comp64    <-->  extended
floating  <-->  floating   (one operand must be extended)
decimal   <-->  extended
```

5. Internal Architecture

The package provides 2-address memory to memory arithmetic operations of the form

```
<op> DST --> DST        and
SRC <op> DST --> DST
```

where DST and SRC are the destination and source operands, respectively. The DST operand is always in the extended format. The conversions have the form:

SRC --> DST

where at least one of SRC and DST is a floating-point format. The package also provides a few support functions in connection with the floating-point error flags and modes.

Extended format results may be coerced to the PRECISION and RANGE of the single or double formats, on an instruction by instruction basis. Then subsequent operations are able to take advantage of the trailing zeros to improve performance. This feature is provided to expedite special-purpose applications such as graphics and is not intended for general use. Only under certain circumstances will it actually obtain a speed advantage, rather than a DISADVANTAGE, since the package is built to do extended arithmetic.

6. External Access

The package is re-entrant, position-independent code, which may be shared in multi-process environments. It is accessed through one entry point, labeled FP68K. Each user process has a static state area consisting of one word of mode bits and error flags, and a two-word halt vector. The package allows for different access to the state word in one-process (Mac) and multi-process (Lisa) environments.

The package preserves all CPU registers across invokations, except that REMAINDER modifies D0. It modifies the CPU condition flags. Except for binary-decimal conversions, it uses little more stack area than is required to save the sixteen 32-bit CPU registers. Since the binary-decimal conversions themselves call the package (to perform multiplies and divides), they use about twice the space of the regular operations.

7. Calling Sequence

A typical invokation of the package will consist of a sequence of four 68K assembly instructions:

```
PEA       <source address>        ;"Push Effective Address"
PEA       <destination address>   ;"Push Effective Address"
MOVE.W    <opword>, -(SP)         ;"Push" operation word
JSR       FP68K                   ;"Call" the package
```

(If FP68K resides in system memory, the JSR may be replaced by an A-line trap opcode.) Other calls will have more or fewer operand addresses to push onto the stack. The opword is the logical OR of two fields, given here in hexadecimal:

"non-extended" operand format, bits 3800:
- 0000 -- x80
- 0800 -- f64
- 1000 -- f32
- 1800 -- ILLEGAL
- 2000 -- int16

```
        2800 -- int32
        3000 -- comp64
        3800 -- ILLEGAL

    arithmetic operation code, bits 001F:
        0000 -- add
        0002 -- subtract
        0004 -- multiply
        0006 -- divide
        0008 -- compare
        000A -- compare and signal invalid if UNORDERED
        000C -- remainder
        000E -- floating, intxx, comp64 --> extended convert
        0010 -- extended --> intXX, comp64, floating convert
        0012 -- square root
        0014 -- round to integer in floating format
        0016 -- truncate to integer in floating format
        0018 -- scale by integer power of 2
        001A -- replace by unbiased exponent
        001C -- classify the floating input
        001E -- ILLEGAL

        0001 -- put state word
        0003 -- get state word
        0005 -- put halt vector
        0007 -- get halt vector
        0009 -- decimal --> floating convert
        000B -- floating --> decimal convert
        000D -- negate
        000F -- absolute value
        0011 -- copy sign
        0013 -- nextafter
        0015 -- set exception
        0017 -- procedure entry protocol
        0019 -- procedure exit protocol
        001B -- test exception
        001D and 001F are ILLEGAL
```

8. Comparisons

In this arithmetic, comparisons require some extra thought. The trichotomy rule of the real number system -- that two numbers are related as LESS, EQUAL, or GREATER -- is violated by the NANs, which compare UNORDERED with everything, even themselves. So it is necessary for floating-point comparisons to use the CPU condition codes in a way that seems surprising at first blush:

RELATION	FLAGS: X	N	Z	V	C
LESS	1	1	0	0	1
EQUAL	0	0	1	0	0
GREATER	0	0	0	0	0
UNORDERED	0	0	0	1	0

This encoding leads to a very convenient mapping between the "floating-point conditional branches" and the CPU conditional branches. In the following table, the '?' refers to UNORDERED. The second column gives the name of the branch macro that provides the "floating branch" (see the "Assembler Support" document).

BRANCH CONDITION		MACRO NOTATION	CPU BRANCH
=		FBEQ	BEQ
<		FBLT	BCS
<, =		FBLE	BLS
>		FBGT	BGT
>, =		FBGE	BGE
?, <		FBULT	BLT
?, <, =		FBULE	BLE
?, >		FBUGT	BHI
?, >, =		FBUGE	BCC
?	(unordered)	FBU	BVS
<, =, >	(ordered)	FBO	BVC
?, <, >	(not equal)	FBNE	BNE
?, =		FBUE	BEQ / BVS
<, >		FBLG	BNE / BVC

Only in the last two instances, are two branches required.

The variant comparison instruction, that signals the invalid operation exception if its operands are UNORDERED, is useful in high-level languages since P754 (and SANE) require that certain UNORDERED comparisons be marked invalid.

Further discussion of the language issues of comparisons may be found in "Comparisons and Branching" by Jerome Coonen.

9. Binary-Decimal Conversions

The package provides conversion functions intended to be used in conjunction with scanners and formatters peculiar to the user environment. For decimal to binary conversions, the input parameters are:

```
address of Pascal decimal structure:
    record
        sgn : 0..1;
        exp : integer;
        sig : string[20]
    end;

address of target floating variable
```

The format (f32, f64, x80) of the target is given in the opword. For binary to decimal conversions, the input paramaters are:

```
address of format structure:
    record
        style : (FloatDecimal, FixedDecimal);
        digits: integer
    end;

address of source floating variable

address of decimal structure:
        sign
        exponent
        ascii string of significant digits
```

The interpretation of the latter format element depends on the style of the conversion. For fixed conversions, the digit count gives the number of fraction digits desired (which may be negative). For float conversions, the digit count gives the number of significant digits desired.

Free format binary --> decimal conversions, which display numbers in the "nicest" format possible within given field width constraints, are supported in software, using the float style of conversion. Nice conversions are handy in applications like accounting spreadsheets where tables of numbers are displayed. See the "Binary-Decimal Conversion" document for details. The SANE interface gives details about the decimal format.

10. The State Area

Each user of the package has three words of static floating-point state information. All accesses to the state should be made through the four state operations. The state consists of:

```
modes and flags word:
        8000 -- unused

        6000 -- rounding direction:
                0000 -- to nearest
                2000 -- toward +INF
                4000 -- toward -INF
                6000 -- toward zero (chop)

        1F00 -- error flags, from high to low order:
                1000 -- inexact result
                0800 -- division by zero
                0400 -- floating overflow
                0200 -- floating underflow
                0100 -- invalid operation


        0080 -- rounding of last result
```

```
                0000 -- not rounded up in magnitude
                0080 -- rounded up in magnitude

        0060 -- precision control:
                0000 -- extended
                0020 -- double
                0040 -- single
                0060 -- ILLEGAL

        001F -- halt enables, correspond to error flags

halt vector:
        32-bit address of alternate exit from package
```

11. Halts

When an error arises for which the corresponding halt is enabled, a trap is taken through the vector in the floating-point state area. The halt routine is called as a Pascal procedure of the form

```
PROCEDURE MyHalt(VAR r: fpRegs; op3, op2, op1: fpPtr; opcode: integer);
where
TYPE
  fpRegs = RECORD BEGIN
             FPRCCR,            { 68000 CCR register }
             FPRD0HI,           { high word of register D0 }
             FPRD0LO            { low word of register D0 }
           END;
  fpPtr = ^Extended;            { but may be pointer to any type }
```

The only way to return to the package from a halt is to initiate a new floating-point operation. There is no way to resume execution of the halted operation.

The state-related operations never halt. The binary-decimal conversions do not halt, though the individual operations they employ (such as multiplication to form 10^N for some integer N) might halt.

12. Other Pseudo-Machines

The package is simple and general enough to be the basis for pseudo-machines with register architectures like the 68881 or the Z8070 or with an evaluation stack like the Intel 8087. What is needed is simply the mechanism to compute addresses in the register file or stack (and check for internal consistency), and the set of functions required to manipulate that isolated data file (e.g. duplicate the top stack element, negate a register).

13. Arithmetic Abuse

The package is designed to be as robust as possible but it is not bullet-proof, since it is specified to modify the stack. If the user passes

illegal addresses, a memory fault may arise when the package attempts to access the operands. And if the user passes the wrong number of address operands, then in general the stack will be irreparably damaged. Operation is undefined if ILLEGAL values are used in the opword parameter.

14. Size and Performance

FP68K is about 4000 bytes long. On a 4mhz system it executes the simplest arithmetic operations in about 0.4ms and requires just over 1.0ms for a full extended multiply. Divide and square root are longer yet.

Comparative timings show that, for double format operations, FP68K is just faster than the AMD 9512 on Lisa and is about twice twice as fast as the Motorola 68341 code. For single format operations, FP68K is about half as fast as the Lisa single-only package, which is just slower than the 9512.

15. Floating-Point at a Glance

Figure 1 at the end of this document illustrates the basic control of flow in the execution of the floating-point package. The figure is followed by a list of observations on the behavior of the package, and of IEEE arithmetic in general.

1. The package has a single entry point.

2. The package has two exit points, one for normal subroutine returns and one for halts through a vector.

3. Three classes of operations are distinguished: arithmetic operations, binary-decimal conversions, and accesses to the state word and halt vector.

4. The not-a-number symbols, NANs, are detected at the start of each operation. Of them, signaling NANs are the most virulent; they always trigger the invalid operation exception. Quiet NANs propagate through operations; a precedence rule determines which is output if two are input.

5. Invalid operations always result in a quiet NAN output. In the case of the discrete types INT16, INT32, COMP64, the output value is all zero bits except for a leading one bit (that is, 100000...). Floating-point NANs contain an error code to indicate their origin (such as 01 for square root of a negative number).

6. When the input operands are unpacked, the special cases 0, FNZ (finite nonzero number), and INF (infinity) are detected. This expedites special cases such as

+INF + FNZ --> +INF

7. When 0 or INF results from a trivial operation like the example above, no further processing is required before the value is packed. All nontrivial floating-point results are subject to precision and range coercion to assure that they fit in the intended destination.

8. Integer results are subject to coercion to detect overflow.

9. Floating-point NAN results are coerced by chopping them to the precision of the destination, and checking that a legitimate value results.

10. Comparisons require special care, since they produce no results but rather modify the CPU condition-code register. Comparisons, even when NANs are involved, must bypass the coercion steps.

Introduction

This is a brief guide to the program FP68K, a software implementation of proposed IEEE standard P754 (Draft 10.0) for binary floating-point arithmetic. This guide is intended to aid a programmer wishing to understand the workings of FP68K.

The code

The software is in the assembly language of the Motorola MC68000, following the Apple "TLA" syntax of the Lisa assembler. FP68K is non-self-modifying, position-independent code. It has no local data area, that is it uses dynamically allocated stack area for all of its temporaries. FP68K is one large subroutine whose single entry point has the name FP68K.

The code is separated into the functionally distinct files:

```
FPDRIVER.TEXT  -- "includes" the other files...
FPEQUS.TEXT    -- defines set of named constants
FPCONTROL.TEXT -- organizes the flow of control
FPUNPACK.TEXT  -- unpack input operands to intermediate format
FPADD.TEXT     -- add and subtract
FPMUL.TEXT     -- multiply
FPDIV.TEXT     -- divide
FPREM.TEXT     -- remainder
FPCMP.TEXT     -- compare
FPSQRT.TEXT    -- square root
FPCVT.TEXT     -- floating <--> floating,integer conversions
FPSLOG.TEXT    -- logb, scalb, and class appendix functions
FPNANS.TEXT    -- handle "Not A Number" symbols
FPCOERCE.TEXT  -- post-normalize, round, check over/underflow...
FPPACK.TEXT    -- pack result to storage format
FPODDS.TEXT    -- non-arithmetic operations
FBD2B.TEXT     -- decimal --> binary conversion
FBB2D.TEXT     -- binary --> decimal conversion
FBPTEN.TEXT    -- computes 10^N for nonnegative integer N
```

As noted, FPDRIVER.TEXT is a short file which simply includes the other files between the ".PROC" header and ".END" trailer.

Assembling FP68K

Assemble the file FPDRIVER.OBJ to produce the FP68K object file.

The one system dependency of FP68K is its access of the floating-point state area, as discussed in the "System Implementor's Guide". Near the top of FPCONTROL.TEXT is the code which pulls the address of the the 3-word state area into register A0. This code will typically require modification when FP68K is moved to a new system. The well-marked comment within FPCONTROL.TEXT indicates the different access schemes systems might use. If the state area

is to be located using a constant defined in a public "include" file, then that file should be included within FPDRIVER.TEXT. See the comment there for details.

Other than its access to the state area, FP68K is intended to system-independent and should not be tailored recklessly.

Control flow

There are three fundamentally distinct classes of operations performed by FP68K: basic arithmetic, binary-decimal conversions, and manipulations of the floating-point state area. The last of these, namely reading and writing the state word and the halt vector, is trivial and needs no explanation beyond the simple code contained in FPODDS.TEXT.

The basic arithmetic operations are illustrated in the flow chart at the end of this note. The chart is marked to distinguish the function of the various files listed above.

The binary-decimal conversions are quite different from the basic operations, and are not described by the basic flow chart. The conversions might better be thought of as subroutines which have been implemented within FP68K as a matter of architectural convenience. The conversions invoke FP68K itself to perform various basic operations like multiply and divide. The binary-decimal algorithms are described in considerable detail in the attached paper "Accurate, Yet Economical Binary-Decimal Conversions" by J. Coonen.

Exponent calculations

FP68K manipulates exponents in a way that might seem surprising at first glance. The P754 extended format, on which all FP68K arithmetic is based, has a 1-bit sign, 15-bit exponent, and a 64-bit significand. However, the actual exponent range is not 0 to 32767 (biased by 16383) as the 15-bit exponent field would suggest. Rather, it is -63 to 32767 because of the presence of tiny denormalized numbers; this is "just a little bit" beyond the stated 15-bit range. (See the attached paper "Underflow and the Denormalized Numbers" by J. Coonen for a discussion of tiny values in P754 arithmetic.)

Because the operations multiply and divide require the addition and subtraction, respectively, of operand exponents in forming their intermediate results, the implementor typically expects to have one extra exponent bit for intermediate calculations. Thus for P754 extended format calculations, there is need for "just a little bit" beyond 16 exponent bits. This elusive 17-th bit is discussed in yet another attached paper, "Are 17 Exponent Bits Too Many?" It is shown there that 16 bits suffice, if care is taken to perform some extra tests in the right places.

On the 68000 it turns out to be convenient to perform exponent calculations in the ADDRESS REGISTERS -- with a full 32 bits. The address registers provide just the right functionality: add, subtract, and compare. And since floating-point arithmetic is computation-intensive on a small data set, only a few of the address registers are actually needed for addresses.

Finally, 16-bit constants like the exponent bias may be added into the 32-bit exponents with a 2-word instruction, since for "address" calculations the constant is first sign-extended out to a full 32 bits.

Bit field encodings

This section describes the various bit fields used by FP68K. Some of them, like the opcode and the state word, are visible to programs invoking FP68K. Others, like the rounding and sign bits, are local to FP68K.

The OPCODE is the last word pushed on the stack before calling FP68K. It is composed of the fields:

3800 -- "non-extended" operand format:
- 0000 -- x80
- 0800 -- f64
- 1000 -- f32
- 1800 -- ILLEGAL
- 2000 -- int16
- 2800 -- int32
- 3000 -- comp64
- 3800 -- ILLEGAL

07E0 -- must be zero

001F -- operation code:
- 0000 -- add
- 0002 -- subtract
- 0004 -- multiply
- 0006 -- divide
- 0008 -- compare
- 000A -- compare (invalid if UNORDERED)
- 000C -- remainder
- 000E -- x80, f64, f32, int16, int32, comp64 --> x80
- 0010 -- x80 --> x80, f64, f32, int16, int32, comp64
- 0012 -- square root (in x80)
- 0014 -- round to integer (in x80)
- 0016 -- truncate to integer (in x80)
- 0018 -- scale by unbiased power of 2
- 001A -- replace by unbiased exponent
- 001C -- classify the floating input
- 001E -- ILLEGAL

- 0001 -- put state word
- 0003 -- get state word
- 0005 -- put halt vector
- 0007 -- get halt vector
- 0009 -- decimal --> floating convert
- 000B -- floating --> decimal convert
- 000D -- negate
- 000F -- absolute value
- 0011 -- copy sign
- 0013 -- nextafter

0015 -- set exception
0017 -- procedure entry protocol
0019 -- procedure exit protocol
001B -- test exception
001D and 001F are ILLEGAL

The STATE word is static data that perseveres across calls to FP68K. As such, it must live in an area outside FP68K, defined by the host system. Typically the state word (and the halt vector, which is a 32-bit address) will live in the system's "per-process data area", perhaps a fixed location in memory or a fixed offset from some reserved address register. Although the STATE word is directly available to the programmer, typical access will be through an intermediate layer of software (available, say, in a Pascal unit) that insulates the programmer from the details of the actual bit encodings. The STATE word is composed of the fields:

8000 -- unused

6000 -- rounding mode:
0000 -- to nearest
2000 -- toward +INF
4000 -- toward -INF
6000 -- toward 0 (chop)

1F00 -- error flags:
1000 -- inexact result
0800 -- division by zero
0400 -- floating overflow
0200 -- floating underflow
0100 -- invalid operation

0080 -- rounding of last result
0000 -- not rounded up in magnitude
0080 -- rounded up in magnitude

0060 -- precision control:
0000 -- extended
0020 -- double
0040 -- single
0060 -- ILLEGAL

001F -- exception halt enables:
(correspond to error flags above)

After preliminary decoding in FPCONTROL.TEXT, the OPCODE is expanded out into the following 16-bit form:

8000 -- nonzero iff result has single precision and range

4000 -- nonzero iff result has double precision and range

3800 -- source operand format:

(same encoding as in OPCODE)

0700 -- destination operand format:
(same encoding as in OPCODE)

0080 -- nonzero iff destination operand is input

0040 -- nonzero iff source operand is input

0020 -- nonzero iff destination operand is output

001E -- operation code:
(same encoding as in OPCODE but with low bit 0)

0001 -- nonzero iff two-address operation

The ROUND BITS, known as "guard", "round", and "sticky" in documentation about P754, are kept in a 16-bit word. Roughly speaking, the guard and round bits are the two bits beyond the least significant bit of the intermediate result, and the sticky bit is the logical Or of all bits thereafter. The sticky bit is necessary to implement the rounding modes of P754. The ROUND BITS are kept as:

8000 -- guard bit
4000 -- round bit
3F00 -- 6 extra round bits
00FF -- sticky bits

The reason for keeping an entire byte of sticky bits lies in the 68000 instruction set. The archetype operation involving the sticky bit is the right-shift. Any time a bit is shifted off the low end of the sticky "byte", it must be logically Or-ed back into sticky. This is done with the 68000 "SCS" instruction, which sets a given byte to all 1s if the carry bit is set, and clears the byte to 0 otherwise. Typically, a bit is shifted off to the right, it is SCS-ed into an auxiliary byte, and that byte is Or-ed into the sticky byte. Although this is the typical use of the sticky byte, the programmer should not assume that the sticky byte is always either all 0s or all 1s. Sometimes, such as in the right shift after a carry-out in ADD/SUB, the logical Or will be omitted since it is known that if a 1 was shifted out of the sticky byte there will necessarily be another 1 left in sticky.

The operands' SIGNS are kept together in a byte as follows:

80 -- source operand sign
40 -- destination operand sign
20 -- Exclusive Or of the two operands' signs
1F -- unused, but not necessarily zero

If there is just one input operand, its sign is in the high order bit. The Exclusive Or is computed just once, at the start of every arithmetic operation. Not only is it required for many common operations (+, -, *, /, REM, CMP), but it is costly in time and space because of the inefficacy of the

68000 bit instructions, so it is worthwhile to implement the code sequence just once.

The CCR (condition code register) bits of the 68000 are modified by every arithmetic operation, though only the compare instructions leave them in a well defined state. A CCR word is maintained by FP68K:

```
FFE0 -- unused, forced to 0
0010 -- X = Extend
0008 -- N = Negative
0004 -- Z = Zero
0002 -- V = Overflow
0001 -- C = Carry
```

The compare operations encode their results as follows:

RELATION	FLAGS: X	N	Z	V	C
LESS	1	1	0	0	1
EQUAL	0	0	1	0	0
GREATER	0	0	0	0	0
UNORDERED	0	0	0	1	0

See the FP68K programmer's manual for the software applications of the CCR field.

Register usage

The key to the speed (such as it is) and compactness of FP68K is that its entire working data set may be held in the 68000 register file. Immediately upon entry, FP68K saves registers D0-D7, A0-A4 on the stack. Then the registers are loaded up as the operation proceeds. Several of the registers have a meaning that perseveres across nearly the entire instruction. The following list gives a rough idea of register usage:

```
D7 hi -- CCR word
D7 lo -- round bits
D6 hi -- opcode word
D6 lo -- error byte (hi) and sign byte (lo)
D5    -- low 32 source (later result) significant bits
D4    -- high 32 source (later result) sig
nificant bits
D3-D0 -- scratch area
```

A

```
 7     -- SP = stack pointer
A6     -- stack link pointer
A5     -- Mac globals poionter
A4     -- source (later result) exponent
A3     -- destination exponent
A2     -- low 32 destination significant bits
A1     -- high 32 destination significant bits
A0     -- pointer to 3-word state area
```


Of course, the arithmetic operations may be viewed as transformations of the register file. Following this view, a set of register maps are included at the end of this note. They are keyed to MILESTONES marked in the source code. The maps indicate register dependencies, and as such should aid in any modification of FP68K. Some maps simply indicate the state of the register file at a given point, and some indicate register use in a routine, such as the widely used right-shift procedure RTSHIFT.

For convenience the maps are printed on onion skin paper; a reference sheet slips under the map to fill in the register mask.

Register D0 is modified by the REMAINDER operation, in which case a partial integer quotient is returned in D0.W.

Stack usage

When called, FP68K assumes that the stack has the form:

```
ADDRESS 3 -- used for decimal format code only
ADDRESS 2 -- source pointer, if any
ADDRESS 1 -- destination pointer
OPCODE    -- one word
RETURN ADDRESS
```

The number of address operands depends on the operation. FP68K then allocates 3 more stack words:

```
COUNT     -- number of bytes in original call frame
HALT ADDRESS
```

This frame is used if a halt is taken. The COUNT field allows the halt handler to simply pop the original operands and return, if desired.

Above this frame, FP68K pushes registers D0-7, A0-6. In the progress of an operation, up to 6 more words of stack may be used. The total stack usage, after the call, is then up to 3 + 32 + 6 = 41 words. The binary-decimal conversions may use twice this much since they invoke FP68K to perform basic arithmetic operations.

Conditional assembly

There are two instances of conditional assembly in FP68K. The pointer to the floating-point state area is loaded into register A0 at the start of FPCONTROL.TEXT. Since the location of this area is system-dependent, conditional assembly is used to locate the field. Of course, this means that the effective address of the state area must be known at assembly time.

Conditional assembly is also used to resolve syntactic inconsistencies between various 68000 assembly language formats. The program counter (PC) relative addressing modes are heavily used in the implementation of jump offset tables within FP68K. A typical use is the instruction sequence:


```
MOVE.W    JMPTAB(D0),D0
JMP       JMPTOP(D0)
```

Here JMPTAB is a table of address offsets from the label JMPTOP, and register D0 contains a word index into JMPTAB. Some assemblers force the programmer to write:

```
MOVE.W    JMPTAB(PC,D0),D0
JMP       JMPTOP(PC,D0)
```

in order to assure PC-relative addressing. However, the Lisa assembler PROHIBITS this syntax, although it produces the desired code. An assembly flag is used to generate whichever of the two formats is suitable for a given compiler.

Pascal enumerated types

Lisa Pascal attempts to encode enumerated types in byte fields, which are then stored as the high byte of the target word. This affects structures like DecForm and Decimal, defined in the Pascal interface (see that document for details). Although the most seriously affected programs are the test drivers, the affected files in the basic package are FBB2D.TEXT and FBD2B.TEXT. Those files contain explicit comments when a byte test is used where an Apple III programmer (for example) might expect a word test.

Value/	Name:				
A000	Open	A030	OSEventAvail	AC61	Random
A001	Close	A031	GetOSEvent	AC62	ForeColor
A002	Read	A032	FlushEvents	AC63	BackColor
A003	Write	A033	VInstall	AC64	ColorBit
A004	Control	A034	VRemove	AC65	GetPixel
A005	Status	A035	OffLine	AC66	StuffHex
A006	KillIO	A036	MoreMasters	AC67	LongMul
A007	GetVolInfo	A037	ReadParam	AC68	FixMul
A008	FileCreate	A038	WriteParam	AC69	FixRatio
A009	FileDelete	A039	ReadDateTime	AC6A	HiWord
A00A	OpenRf	A03A	SetDateTime	AC6B	LoWord
A00B	Rename	A03B	Delay	AC6C	FixRound
A00C	GetFileInfo	A03C	CmpString	AC6D	InitPort
A00D	SetFileInfo	A03D	DrvrInstall	AC6E	InitGraf
A00E	UnmountVol	A03E	DrvrRemove	AC6F	OpenPort
A00F	MountVol	A03F	InitUtil	AC70	LocalToGlobal
A010	FileAllocate	A040	ResrvMem	AC71	GlobalToLocal
A011	GetEOF	A041	SetFilLock	AC72	GrafDevice
A012	SetEOF	A042	RstFilLock	AC73	SetPort
A013	FlushVol	A043	SetFilType	AC74	GetPort
A014	GetVol	A044	SetFPos	AC75	SetPortBits
A015	SetVol	A045	FlushFil	AC76	PortSize
A016	FInitQueue	A046	GetTrapAddress	AC77	MovePortTo
A017	Eject	A047	SetTrapAddress	AC78	SetOrigin
A018	GetFPos	A048	PtrZone	AC79	SetClip
A019	InitZone	A049	HPurge	AC7A	GetClip
A01A	GetZone	A04A	HNoPurge	AC7B	ClipRect
A01B	SetZone	A04B	SetGrowZone	AC7C	BackPat
A01C	FreeMem	A04C	CompactMem	AC7D	ClosePort
A01D	MaxMem	A04D	PurgeMem	AC7E	AddPt
A01E	NewPtr	A04E	AddDrive	AC7F	SubPt
A01F	DisposePtr	A04F	InstallRDrivers	AC80	SetPt
A020	SetPtrSize	AC50	InitCursor	AC81	EqualPt
A021	GetPtrSize	AC51	SetCursor	AC82	StdText
A022	NWHandle	AC52	HideCursor	AC83	DrawChar
A023	DsposeHandle	AC53	ShowCursor	AC84	DrawString
A024	SetHandleSize	AC54	UprString	AC85	DrawText
A025	GetHandleSize	AC55	ShieldCursor	AC86	TextWidth
A026	HandleZone	AC56	ObscureCursor	AC87	TextFont
A027	ReAllocHandle	AC57	SetApplBase	AC88	TextFace
A028	RecoverHandle	AC58	BitAnd	AC89	TextMode
A029	HLock	AC59	BitXor	AC8A	TextSize
A02A	HUnlock	AC5A	BitNot	AC8B	GetFontInfo
A02B	EmptyHandle	AC5B	BitOr	AC8C	StringWidth
A02C	InitApplZone	AC5C	BitShift	AC8D	CharWidth
A02D	SetApplLimit	AC5D	BitTst	AC8E	SpaceExtra
A02E	BlockMove	AC5E	BitSet	AC90	StdLine
A02F	PostEvent	AC5F	BitClr	AC91	LineTo

Value/	Name:				
AC92	Line	ACC5	StdPoly	ACF6	DrawPicture
AC93	MoveTo	ACC6	FramePoly	ACF8	ScalePt
AC94	Moov	ACC7	PaintPoly	ACF9	MapPt
AC96	HidePen	ACC8	ErasePoly	ACFA	MapRect
AC97	ShowPen	ACC9	InvertPoly	ACFB	MapRgn
AC98	GetPenState	ACCA	FillPoly	ACFC	MapPoly
AC99	SetPenState	ACCB	OpenPoly	ACFE	InitFonts
AC9A	GetPen	ACCC	ClosePoly	ACFF	GetFontName
AC9B	PenSize	ACCD	KillPoly	AD00	GetFNum
AC9C	PenMode	ACCE	OffsetPoly	AD01	FMSwapFont
AC9D	PenPat	ACCF	PackBits	AD02	RealFont
AC9E	PenNormal	ACD0	UnPackBits	AD03	SetFontLock
ACA0	StdRect	ACD1	StdRgn	AD04	DrawGrowIcon
ACA1	FrameRect	ACD2	FrameRgn	AD05	DragGrayRgn
ACA2	PaintRect	ACD3	PaintRgn	AD06	NewString
ACA3	EraseRect	ACD4	EraseRgn	AD07	SetString
ACA4	InvertRect	ACD5	InvertRgn	AD08	ShowHide
ACA5	FillRect	ACD6	FillRgn	AD09	CalcVis
ACA6	EqualRect	ACD8	NewRgn	AD0A	CalcVisBehind
ACA7	SetRect	ACD9	DisposeRgn	AD0B	ClipAbove
ACA8	OffsetRect	ACDA	OpenRgn	AD0C	PaintOne
ACA9	InsetRect	ACDB	CloseRgn	AD0D	PaintBehind
ACAA	SectRect	ACDC	CopyRgn	AD0E	SaveOld
ACAB	UnionRect	ACDD	SetEmptyRgn	AD0F	DrawNew
ACAC	Pt2Rect	ACDE	SetRectRgn	AD10	GetWMgrPort
ACAD	PtInRect	ACDF	RectRgn	AD11	CheckUpDate
ACAE	EmptyRect	ACE0	OffsetRgn	AD12	InitWindows
ACAF	StdRRect	ACE1	InsetRgn	AD13	NewWindow
ACB0	FrameRoundRect	ACE2	EmptyRgn	AD14	DisposeWindow
ACB1	PaintRoundRect	ACE3	EqualRgn	AD15	ShowWindow
ACB2	EraseRoundRect	ACE4	SectRgn	AD16	HideWindow
ACB3	InvertRoundRect	ACE5	UnionRgn	AD17	GetWRefCon
ACB4	FillRoundRect	ACE6	DiffRgn	AD18	SetWRefCon
ACB6	StdOval	ACE7	XOrRgn	AD19	GetWTitle
ACB7	FrameOval	ACE8	PtInRgn	AD1A	SetWTitle
ACB8	PaintOval	ACE9	RectInRg	AD1B	MoveWindow
ACB9	EraseOval	ACEA	SetStdProcs	AD1C	HiliteWindow
ACBA	InvertOval	ACEB	StdBits	AD1D	SizeWindow
ACBB	FillOval	ACEC	CopyBits	AD1E	TrackGoAway
ACBC	SlopeFromAngle	ACED	StdTxMeasure	AD1F	SelectWindow
ACBD	StdArc	ACEE	StdGetPic	AD20	BringToFront
ACBE	FrameArc	ACEF	ScrollRect	AD21	SendBehind
ACBF	PaintArc	ACF0	StdPutPic	AD22	BeginUpdate
ACC0	EraseArc	ACF1	StdComment	AD23	EndUpdate
ACC1	InvertArc	ACF2	PicComment	AD24	FrontWindow
ACC2	FillArc	ACF3	OpenPicture	AD25	DragWindow
ACC3	PtToAngle	ACF4	ClosePicture	AD26	DragTheRgn
ACC4	AngleFromSlope	ACF5	KillPicture	AD27	InvalRgn

Name:	Value/ Fields:				
OpenPort	AC6F	ReAllocHandle	A027	SetOrigin	AC78
OpenResFile	AD97	RecoverHandle	A028	SetPenState	AC99
OpenRf	A00A	RectInRg	ACE9	SetPort	AC73
OpenRgn	ACDA	RectRgn	ACDF	SetPortBits	AC75
OSEventAvail	A030	ReleaseResource	ADA3	SetPt	AC80
Pack0	ADE7	Rename	A00B	SetPtrSize	A020
Pack1	ADE8	ResError	ADAF	SetRect	ACA7
Pack2	ADE9	ResrvMem	A040	SetRectRgn	ACDE
Pack3	ADEA	RmveReference	ADAE	SetResAttrs	ADA7
Pack4	ADEB	RmveResource	ADAD	SetResFileAttrs	ADF7
Pack5	ADEC	RsrcZoneInit	AD96	SetResInfo	ADA9
Pack6	ADED	RstFilLock	A042	SetResLoad	AD9B
Pack7	ADEE	SaveOld	AD0E	SetResPurge	AD93
PackBits	ACCF	ScalePt	ACF8	SetStdProcs	ACEA
PaintArc	ACBF	ScrollRect	ACEF	SetString	AD07
PaintBehind	AD0D	SectRect	ACAA	SetTrapAddress	A047
PaintOne	AD0C	SectRgn	ACE4	SetVol	A015
PaintOval	ACB8	SelectWindow	AD1F	SetWindowPic	AD2E
PaintPoly	ACC7	SendBehind	AD21	SetWRefCon	AD18
PaintRect	ACA2	SetApplBase	AC57	SetWTitle	AD1A
PaintRgn	ACD3	SetApplLimit	A02D	SetZone	A01B
PaintRoundRect	ACB1	SetClip	AC79	ShieldCursor	AC55
ParamText	AD8B	SetCRefCon	AD5B	ShowControl	AD57
PenMode	AC9C	SetCTitle	AD5F	ShowCursor	AC53
PenNormal	AC9E	SetCtlAction	AD6B	ShowHide	AD08
PenPat	AC9D	SetCtlMax	AD65	ShowPen	AC97
PenSize	AC9B	SetCtlMin	AD64	ShowWindow	AD15
PicComment	ACF2	SetCtlValue	AD63	SizeControl	AD5C
PinRect	AD4E	SetCursor	AC51	SizeRsrc	ADA5
PlotIcon	AD4B	SetDateTime	A03A	SizeWindow	AD1D
PortSize	AC76	SetDItem	AD8E	SlopeFromAngle	ACBC
PostEvent	A02F	SetEmptyRgn	ACDD	SpaceExtra	AC8E
Pt2Rect	ACAC	SetEOF	A012	Status	A005
PtInRect	ACAD	SetFileInfo	A00D	StdArc	ACBD
PtInRgn	ACE8	SetFilLock	A041	StdBits	ACEB
PtrAndHand	ADEF	SetFilType	A043	StdComment	ACF1
PtrToHand	ADE3	SetFontLock	AD03	StdGetPic	ACEE
PtrToXHand	ADE2	SetFPos	A044	StdLine	AC90
PtrZone	A048	SetGrowZone	A04B	StdOval	ACB6
PtToAngle	ACC3	SetHandleSize	A024	StdPoly	ACC5
PurgeMem	A04D	SetItem	AD47	StdPutPic	ACF0
PutIcon	ADCA	SetItemIcon	AD40	StdRect	ACA0
PutScrap	ADFE	SetItemMark	AD44	StdRgn	ACD1
Random	AC61	SetItemStyle	AD42	StdRRect	ACAF
Read	A002	SetIText	AD7E	StdText	AC82
ReadDateTime	A039	SetIText	AD8F	StdTxMeasure	ACED
ReadParam	A037	SetMenuBar	AD3C	StillDown	AD73
RealFont	AD02	SetMenuFlash	AD4A	StopAlert	AD86

Name:	Value/	Fields:	
StringWidth	AC8C	UprString	AC54
StuffHex	AC66	UseResFile	AD98
SubPt	AC7F	ValidRect	AD2A
SystemBeep	ADC8	ValidRgn	AD29
SystemClick	ADB3	VInstall	A033
SystemEdit	ADC2	VRemove	A034
SystemError	ADC9	WaitMouseUp	AD77
SystemEvent	ADB2	Write	A003
SystemMenu	ADB5	WriteParam	A038
SystemTask	ADB4	WriteResource	ADB0
TEActivate	ADD8	XOrRgn	ACE7
TECalText	ADD0	ZeroScrap	ADFC
TEClick	ADD4		
TECopy	ADD5		
TECut	ADD6		
TEDeactivate	ADD9		
TEDelete	ADD7		
TEDispose	ADCD		
TeGetText	ADCB		
TEIdle	ADDA		
TEInit	ADCC		
TEInsert	ADDE		
TEKey	ADDC		
TENew	ADD2		
TEPaste	ADDB		
TEScroll	ADDD		
TESetJust	ADDF		
TESetSelect	ADD1		
TESetText	ADCF		
TestControl	AD66		
TEUpdate	ADD3		
TextBox	ADCE		
TextFace	AC88		
TextFont	AC87		
TextMode	AC89		
TextSize	AC8A		
TextWidth	AC86		
TickCount	AD75		
TrackControl	AD68		
TrackGoAway	AD1E		
UnionRect	ACAB		
UnionRgn	ACE5		
UniqueID	ADC1		
UnloadScrap	ADFA		
UnLoadSeg	ADF1		
UnmountVol	A00E		
UnPackBits	ACD0		
UpdateResFile	AD99		

Selection: Value/Trap: equals A000
through Value/Trap: equals FFFF

Name:	Value/	Fields:			
AddDrive	A04E	CopyRgn	ACDC	EraseArc	ACC0
AddPt	AC7E	CouldAlert	AD89	EraseOval	ACB9
AddReference	ADAC	CouldDialog	AD79	ErasePoly	ACC8
AddResMenu	AD4D	CountMItems	AD50	EraseRect	ACA3
AddResource	ADAB	CountResources	AD9C	EraseRgn	ACD4
Alert	AD85	CountTypes	AD9E	EraseRoundRect	ACB2
AngleFromSlope	ACC4	CreateResFile	ADB1	ErrorSound	AD8C
AppendMenu	AD33	CurResFile	AD94	EventAvail	AD71
BackColor	AC63	Delay	A03B	ExitToShell	ADF4
BackPat	AC7C	DeleteMenu	AD36	FileAllocate	A010
BeginUpdate	AD22	DeltaPoint	AD4F	FileCreate	A008
BitAnd	AC58	DeQueue	AD6E	FileDelete	A009
BitClr	AC5F	DetachResouce	AD92	FillArc	ACC2
BitNot	AC5A	DialogSelect	AD80	FillOval	ACBB
BitOr	AC5B	DiffRgn	ACE6	FillPoly	ACCA
BitSet	AC5E	DisableItem	AD3A	FillRect	ACA5
BitShift	AC5C	DisposeControl	AD55	FillRgn	ACD6
BitTst	AC5D	DisposeDialog	AD83	FillRoundRect	ACB4
BitXor	AC59	DisposeMenu	AD32	FindControl	AD6C
BlockMove	A02E	DisposePtr	A01F	FindWindow	AD2C
BringToFront	AD20	DisposeRgn	ACD9	FInitQueue	A016
Button	AD74	DisposeWindow	AD14	FixMul	AC68
CalcMenuSize	AD48	DragControl	AD67	FixRatio	AC69
CalcVis	AD09	DragGrayRgn	AD05	FixRound	AC6C
CalcVisBehind	AD0A	DragTheRgn	AD26	FlashMenuBar	AD4C
CautionAlert	AD88	DragWindow	AD25	FlushEvents	A032
Chain	ADF3	DrawChar	AC83	FlushFil	A045
ChangedResData	ADAA	DrawControls	AD69	FlushVol	A013
CharWidth	AC8D	DrawDialog	AD81	FMSwapFont	AD01
CheckItem	AD45	DrawGrowIcon	AD04	ForeColor	AC62
CheckUpDate	AD11	DrawMenuBar	AD37	FrameArc	ACBE
ClearMenuBar	AD34	DrawNew	AD0F	FrameOval	ACB7
ClipAbove	AD0B	DrawPicture	ACF6	FramePoly	ACC6
ClipRect	AC7B	DrawString	AC84	FrameRect	ACA1
Close	A001	DrawText	AC85	FrameRgn	ACD2
CloseDeskAcc	ADB7	DrvrInstall	A03D	FrameRoundRect	ACB0
CloseDialog	AD82	DrvrRemove	A03E	FreeAlert	AD8A
ClosePicture	ACF4	DsposeHandle	A023	FreeDialog	AD7A
ClosePoly	ACCC	Eject	A017	FreeMem	A01C
ClosePort	AC7D	EmptyHandle	A02B	FrontWindow	AD24
CloseResFile	AD9A	EmptyRect	ACAE	GetAppParms	ADF5
CloseRgn	ACDB	EmptyRgn	ACE2	GetClip	AC7A
CloseWindow	AD2D	EnableItem	AD39	GetCRefCon	AD5A
CmpString	A03C	EndUpdate	AD23	GetCTitle	AD5E
ColorBit	AC64	EnQueue	AD6F	GetCtlAction	AD6A
CompactMem	A04C	EqualPt	AC81	GetCtlMax	AD62
Control	A004	EqualRect	ACA6	GetCtlMin	AD61
CopyBits	ACEC	EqualRgn	ACE3	GetCtlValue	AD60

Name:	Value/ Fields:				
GetCursor	ADB9	GetWTitle	AD19	IsDialogEvent	AD7F
GetDItem	AD8D	GetZone	A01A	KillControls	AD56
GetEOF	A011	GlobalToLocal	AC71	KillIO	A006
GetFileInfo	A00C	GrafDevice	AC72	KillPicture	ACF5
GetFNum	AD00	GrowWindow	AD2B	KillPoly	ACCD
GetFontInfo	AC8B	HandAndHand	ADE4	Launch	ADF2
GetFontName	ACFF	HandleZone	A026	Line	AC92
GetFPos	A018	HandToHand	ADE1	LineTo	AC91
GetHandleSize	A025	HideControl	AD58	LoadResource	ADA2
GetIcon	ADBB	HideCursor	AC52	LoadScrap	ADFB
GetIndResource	AD9D	HidePen	AC96	LoadSeg	ADF0
GetIndType	AD9F	HideWindow	AD16	LocalToGlobal	AC70
GetItem	AD46	HiliteControl	AD5D	LongMul	AC67
GetItemIcon	AD3F	HiliteMenu	AD38	LoWord	AC6B
GetItemMark	AD43	HiliteWindow	AD1C	MapPoly	ACFC
GetItemStyle	AD41	HiWord	AC6A	MapPt	ACF9
GetIText	AD90	HLock	A029	MapRect	ACFA
GetKeys	AD76	HNoPurge	A04A	MapRgn	ACFB
GetMenu	ADBF	HomeResFile	ADA4	MaxMem	A01D
GetMenuBar	AD3B	HPurge	A049	MenuKey	AD3E
GetMHandle	AD49	HUnlock	A02A	MenuSelect	AD3D
GetMouse	AD72	InfoScrap	ADF9	ModalDialog	AD91
GetNamedResourc	ADA1	InitApplZone	A02C	Moov	AC94
GetNewControl	ADBE	InitCursor	AC50	MoreMasters	A036
GetNewDialog	AD7C	InitDialogs	AD7B	MountVol	A00F
GetNewMBar	ADC0	InitFonts	ACFE	MoveControl	AD59
GetNewWindow	ADBD	InitGraf	AC6E	MovePortTo	AC77
GetNextEvent	AD70	InitMath	ADE6	MoveTo	AC93
GetOSEvent	A031	InitMenus	AD30	MoveWindow	AD1B
GetPattern	ADB8	InitPack	ADE5	Munger	ADE0
GetPen	AC9A	InitPort	AC6D	NewControl	AD54
GetPenState	AC98	InitResources	AD95	NewDialog	AD7D
GetPicture	ADBC	InitUtil	A03F	NewMenu	AD31
GetPixel	AC65	InitWindows	AD12	NewPtr	A01E
GetPort	AC74	InitZone	A019	NewRgn	ACD8
GetPtrSize	A021	InsertMenu	AD35	NewString	AD06
GetResAttrs	ADA6	InsertResMenu	AD51	NewWindow	AD13
GetResFileAttrs	ADF6	InsetRect	ACA9	NoteAlert	AD87
GetResInfo	ADA8	InsetRgn	ACE1	NWHandle	A022
GetResource	ADA0	InstallRDrivers	A04F	ObscureCursor	AC56
GetScrap	ADFD	InvalRect	AD28	OffLine	A035
GetString	ADBA	InvalRgn	AD27	OffsetPoly	ACCE
GetTrapAddress	A046	InvertArc	ACC1	OffsetRect	ACA8
GetVol	A014	InvertOval	ACBA	OffsetRgn	ACE0
GetVolInfo	A007	InvertPoly	ACC9	Open	A000
GetWindowPic	AD2F	InvertRect	ACA4	OpenDeskAcc	ADB6
GetWMgrPort	AD10	InvertRgn	ACD5	OpenPicture	ACF3
GetWRefCon	AD17	InvertRoundRect	ACB3	OpenPoly	ACCB

Value/	Name:	Fields:			
AD28	InvalRect	AD5A	GetCRefCon	AD8D	GetDItem
AD29	ValidRgn	AD5B	SetCRefCon	AD8E	SetDItem
AD2A	ValidRect	AD5C	SizeControl	AD8F	SetIText
AD2B	GrowWindow	AD5D	HiliteControl	AD90	GetIText
AD2C	FindWindow	AD5E	GetCTitle	AD91	ModalDialog
AD2D	CloseWindow	AD5F	SetCTitle	AD92	DetachResouce
AD2E	SetWindowPic	AD60	GetCtlValue	AD93	SetResPurge
AD2F	GetWindowPic	AD61	GetCtlMin	AD94	CurResFile
AD30	InitMenus	AD62	GetCtlMax	AD95	InitResources
AD31	NewMenu	AD63	SetCtlValue	AD96	RsrcZoneInit
AD32	DisposeMenu	AD64	SetCtlMin	AD97	OpenResFile
AD33	AppendMenu	AD65	SetCtlMax	AD98	UseResFile
AD34	ClearMenuBar	AD66	TestControl	AD99	UpdateResFile
AD35	InsertMenu	AD67	DragControl	AD9A	CloseResFile
AD36	DeleteMenu	AD68	TrackControl	AD9B	SetResLoad
AD37	DrawMenuBar	AD69	DrawControls	AD9C	CountResources
AD38	HiliteMenu	AD6A	GetCtlAction	AD9D	GetIndResource
AD39	EnableItem	AD6B	SetCtlAction	AD9E	CountTypes
AD3A	DisableItem	AD6C	FindControl	AD9F	GetIndType
AD3B	GetMenuBar	AD6E	DeQueue	ADA0	GetResource
AD3C	SetMenuBar	AD6F	EnQueue	ADA1	GetNamedResourc
AD3D	MenuSelect	AD70	GetNextEvent	ADA2	LoadResource
AD3E	MenuKey	AD71	EventAvail	ADA3	ReleaseResource
AD3F	GetItemIcon	AD72	GetMouse	ADA4	HomeResFile
AD40	SetItemIcon	AD73	StillDown	ADA5	SizeRsrc
AD41	GetItemStyle	AD74	Button	ADA6	GetResAttrs
AD42	SetItemStyle	AD75	TickCount	ADA7	SetResAttrs
AD43	GetItemMark	AD76	GetKeys	ADA8	GetResInfo
AD44	SetItemMark	AD77	WaitMouseUp	ADA9	SetResInfo
AD45	CheckItem	AD79	CouldDialog	ADAA	ChangedResData
AD46	GetItem	AD7A	FreeDialog	ADAB	AddResource
AD47	SetItem	AD7B	InitDialogs	ADAC	AddReference
AD48	CalcMenuSize	AD7C	GetNewDialog	ADAD	RmveResource
AD49	GetMHandle	AD7D	NewDialog	ADAE	RmveReference
AD4A	SetMenuFlash	AD7E	SetIText	ADAF	ResError
AD4B	PlotIcon	AD7F	IsDialogEvent	ADB0	WriteResource
AD4C	FlashMenuBar	AD80	DialogSelect	ADB1	CreateResFile
AD4D	AddResMenu	AD81	DrawDialog	ADB2	SystemEvent
AD4E	PinRect	AD82	CloseDialog	ADB3	SystemClick
AD4F	DeltaPoint	AD83	DisposeDialog	ADB4	SystemTask
AD50	CountMItems	AD85	Alert	ADB5	SystemMenu
AD51	InsertResMenu	AD86	StopAlert	ADB6	OpenDeskAcc
AD54	NewControl	AD87	NoteAlert	ADB7	CloseDeskAcc
AD55	DisposeControl	AD88	CautionAlert	ADB8	GetPattern
AD56	KillControls	AD89	CouldAlert	ADB9	GetCursor
AD57	ShowControl	AD8A	FreeAlert	ADBA	GetString
AD58	HideControl	AD8B	ParamText	ADBB	GetIcon
AD59	MoveControl	AD8C	ErrorSound	ADBC	GetPicture

Value/	Name:	Fields:
ADBD	GetNewWindow	
ADBE	GetNewControl	
ADBF	GetMenu	
ADC0	GetNewMBar	
ADC1	UniqueID	
ADC2	SystemEdit	
ADC8	SystemBeep	
ADC9	SystemError	
ADCA	PutIcon	
ADCB	TeGetText	
ADCC	TEInit	
ADCD	TEDispose	
ADCE	TextBox	
ADCF	TESetText	
ADD0	TECalText	
ADD1	TESetSelect	
ADD2	TENew	
ADD3	TEUpdate	
ADD4	TEClick	
ADD5	TECopy	
ADD6	TECut	
ADD7	TEDelete	
ADD8	TEActivate	
ADD9	TEDeactivate	
ADDA	TEIdle	
ADDB	TEPaste	
ADDC	TEKey	
ADDD	TEScroll	
ADDE	TEInsert	
ADDF	TESetJust	
ADE0	Munger	
ADE1	HandToHand	
ADE2	PtrToXHand	
ADE3	PtrToHand	
ADE4	HandAndHand	
ADE5	InitPack	
ADE6	InitMath	
ADE7	Pack0	
ADE8	Pack1	
ADE9	Pack2	
ADEA	Pack3	
ADEB	Pack4	
ADEC	Pack5	
ADED	Pack6	
ADEE	Pack7	
ADEF	PtrAndHand	
ADF0	LoadSeg	
ADF1	UnLoadSeg	
ADF2	Launch	
ADF3	Chain	
ADF4	ExitToShell	
ADF5	GetAppParms	
ADF6	GetResFileAttrs	
ADF7	SetResFileAttrs	
ADF9	InfoScrap	
ADFA	UnloadScrap	
ADFB	LoadScrap	
ADFC	ZeroScrap	
ADFD	GetScrap	
ADFE	PutScrap	

ROM 7.0 MacsBug Summary

To use MacsBug, include it on your Mac diskette. The system will say 'MacsBug installed' when the diskette is booted. You may also include the Disassembler in the same manner.

The Mac's modem port should be connected to another computer or terminal running at 9600 baud, no parity. Press the interrupt switch after booting the disk. The mouse should freeze and no error message should appear. On the terminal, a register dump should appear, and an asterisk '*' prompt.

Commands available:

DM	Display Memory	DM 100 100	DM RA7,-10 20
SM	Set Memory	SM 0 1 2 3 4 5 6	SM 0 'ABCDE'
D#	Display/Set data register #		D0 0000FFFF
A#	Display/Set address register #		A0
SR	Display/Set status register		
PC	Display/Set program counter		
US	Display/Set user stack	(normally A7)	
SS	Display/Set supervisor stack	(normally A7)	
BR	Display/Set break points (up to eight)		BR BR 4DD0 552A BR CLEAR
A	Display all address registers		
D	Display all data registers		
TD	Display all registers		
CV	Convert between base 10 and 16 (all arithmetic is 32 bit)		CV $FDEF CV &65536
	To do hexidecimal addition, use CV $num1,num2		
	To do hexidecimal subtraction, use CV $num1,-num2		
	To do hexidecimal negation, use CV $-num1		
G	Go	(continue)	G
		(start at 44D0)	G 44D0
		(continue until PC = 55EA)	G TILL 55EA
T	Trace		T 17
AT	Trace Traps	(traces all traps)	AT
		(trace GetNextEvent)	AT 170
		(trace all Bit Traps)	AT 158 15F
		(trace GetNextEvent in code block at 5000 - 53FF)	AT 170 5000 53FF
		(trace all Bit Traps in code block at 5000 - 53FF)	AT 158 15F 5000 53FF

AB	Break Traps	same as AT, but breaks before executing trap	
HD	Handle Display	lists all allocated handles	HD
AD	Data Break		AD 158 15F 5000 53FF

A simple checksum is calculated for the specified memory range. As each Trap is encountered, the checksum is recalculated. If the checksum differs, the debugger breaks. This call must be made with all four arguments.

AX	Cancel Break	AX

Clears the current AT, AB, AH, AC, or HS command

Disassembler Calls

IL	List lists the next 20 instructions	IL 4DD0 IL
ID	List One lists one instruction	ID 4DD0 ID

Debugger Notation

/	Command Separator	SM PC 4E71 / G
.	Last Address (for DM, SM, IL, ID)	DM . 100
,	Offset	DM .,100 100
RA#	Address Register	DM RA7,-10 20
RD#	Data Register	SM RA0, RD2

Advanced Debugger Calls

AH	Heap Break	AH 158 15F

A heap check is made as each specified Trap is encountered.
If $1A3E8 = 0 then the applzone is checked. (default)
If $1A3E8 <> 0 then SysZone is checked. The trap range must be greater than $2E.

An error returns: Bad Heap at A1 A2 where:
A1 = the previous block pointer
A2 = the bad block pointer

HC Heap Check HC

This call checks the heap as described in AH. An error is returned if any of the following conditions are true:

The block size is past the top of memory
The block size is odd
For tree blocks, the next link is negative or past the top of memory
For tree blocks, the previous link is negative or past the top of memory
For rel. blocks, the back pointer is odd
The heap base + back pointer is past the top of memory
The heap base + back pointer do not point to the right master pointer.

HS Heap Scramble HS

If the traps NewPtr, SetPtrSize, NewHandle, SetHandleSize, HandleZone or ReAllocHandle are encountered, the heap is scrambled before executing the trap. It also preforms a heap check before scrambling on all traps > 30.

MR Magic Return MR

This assumes the first word on the stack is a return address generated by a BSR or JSR. It substitutes a break point for the return address. The execution continues until a break occurs. Then, SR is restored.

This is not nestable. All other break point commands are still active.

Known Problems

It is a good idea to initialize DM and IL. DM 0, and ID PC, for example.

DM RA5, as example, intermittently generates an address error. To fix, explicitly type the address in register.

SM PC 4E71, for example, makes the system respond unreliably if a trap or breakpoint was set at that location.

AT, as example, returns a Line 1111 exception. To fix, reboot.

Pascal Program Debug Strategy

Use DM to determine where the program is in memory. Seven letters of each procedure and function will appear in the ASCII columns. (The first letter has its high bit set.) The user program usually starts about 4DD0. The mainline procedures and functions are first, followed by the units and external procedures and functions in the order that you link them. Each procedure or function name suceeds the procedure or function code. To make life easier, link your own units before linking ToolTraps, MemTraps and MacPasLib.

If the program doesn't appear to work at all, find the address of the start of the program. It will be immediately after the name of the last procedure or function.

If you disassemble at that address, you will see a LINK instruction for A6 and a LINK instruction for A5. These address registers are used by Pascal to locate all variables and procedural parameters. Global variables are referenced negatively off A5. Local variables are referenced negatively off A6. Procedural parameters are referenced with a positive offset from A6.

ToolBox calls and other calls to unit-resident procedures and functions are made through JSR *+addr instructions. The table ToolTraps is linked to your program, and contains all of the actual calls themselves.

If you are writing a Pascal program that uses the ToolBox, the first thing you probably do is:

```
InitGraf (@thePort);
```

This assembles into:

```
        LEA     $FF1E(A5),A0
        MOVE.L  A0,A7
        JSR     *+addr
```

You should see this shortly after the LINK instructions. This establishes the beginning of your program.

To find out where this program fails, interrupt the Finder. Set G TILL addr where addr is the beginning of your program. Restart your program. If the program breaks sucessfully at that point, continue to use G TILL to selectly execute your program until it fails more spectacularly, or locks up.

You will find that the 68000 code that the Pascal compiler produces is reasonably readable, and that the compiler produces smart code.

To complement the debugger information, you may want to add debugging code in your program itself. One easy way to do so is to do WRITE s or WRITELN s to the .BOUT port. This information will appear on your debugging terminal. The code fragments required look like:

```
VAR debug : TEXT;

REWRITE (debug, '.BOUT');

WRITELN (debug, chr(10) {linefeed}, 'This is a test', 12345);
```

WRITE and WRITELN support strings, chars, packed array of chars, integers, and booleans.

The MacPaint Document Format

by Bill Atkinson

MacPaint documents are easy to read and write, and have become a standard interchange format for full-page bitmap images on Macintosh. Their internal format is described here to aid program developers in generating and reading MacPaint documents.

MacPaint documents use only the data fork of the file system; the resource fork is not used and may be ignored. The data fork contains a 512 byte header and then the compressed data representing a single bitmap of 576 pixels wide by 720 pixels tall. At 72 pixels per inch, this bitmap occupies the full 8 by 10 inch printable area of the Imagewriter printer page.

HEADER:

The first 512 bytes of the document form a header with a 4 byte version number (default = 2), then 38*8 = 304 bytes of patterns, then 204 unused bytes reserved for future expansion. If the version number is zero, the rest of the header block is ignored and default patterns are used, so programs generating MacPaint documents can simply write out 512 bytes of zero as the document header. Most programs which read MacPaint documents can simply skip over the header when reading.

BITMAP:

Following the header are 720 compressed scanlines of data which form the 576 wide by 720 tall bitmap. Without compression, this bitmap would occupy 51840 bytes and chew up disk space pretty fast; typical MacPaint documents compress to about 10 Kbytes using the PackBits procedure in the Macintosh ROM to compress runs of equal bytes within each scanline. The bitmap part of a MacPaint document is simply 720 times the output of PackBits with 72 bytes input.

READING SAMPLE:

```
CONST srcBlocks = 2;  { at least 2, bigger makes it faster }
      srcSize = 1024; { 512 * srcBlocks }
TYPE  diskBlock = PACKED ARRAY[1..512] OF QDByte;
VAR   srcBuf: ARRAY[1..srcBlocks] OF diskBlock;
      srcPtr,dstPtr: QDPtr;

{ skip the header }
ReadData(srcFile,@srcBuf,512);

{ prime srcBuf }
ReadData(srcFile,@srcBuf,srcSize);

{ unpack each scanline into dstBits, reading more source as needed }
srcPtr := @srcBuf;
dstPtr := dstBits.baseAddr;
FOR scanLine := 1 to 720 DO
   BEGIN
      UnPackBits(srcPtr,dstPtr,72);              { bumps both ptrs }
      { time to read next chunk of packed source ? }
      IF ORD(srcPtr) > ORD(@srcBuf) + srcSize - 512 THEN
         BEGIN
            srcBuf[1] := srcBuf[srcBlocks];      { move up last block }
            ReadData(srcFile,@srcBuf[2],srcSize-512);
            srcPtr := Pointer(ORD(srcPtr) - srcSize + 512);
         END;
   END;
```

WRITING SAMPLE:

To write out a 576 by 720 bitmap which is contained in memory, the following fragment of code could be used:

```
TYPE  diskBlock =      PACKED ARRAY[1..512] OF QDByte;
VAR   srcPtr,dstPtr:   QDPtr;
      dstBuf:          diskBlock;
      dstBytes:        INTEGER;

{ write the header, all zeros }
FOR i := 1 to 512 DO dstBuf[i] := 0;
WriteData(dstFile,@dstBuf,512);

{ Compress each scanline and write it }
srcPtr := srcBits.baseAddr;
FOR scanLine := 1 to 720 DO
   BEGIN
      dstPtr := @dstBuf;
      PackBits(srcPtr,dstPtr,72);                 { bumps both ptrs }
      dstBytes := ORD(dstPtr) - ORD(@dstBuf);     { calc packed size }
      WriteData(dstFile,@dstBuf,dstBytes);        { write packed data }
   END;
```

December 8, 1983

TO MACINTOSH SOFTWARE DEVELOPERS:

We hope that this letter finds all of you busy at work on your application for Macintosh. We at Apple are very excited to have you as a software developer and look forward to seeing your product on Macintosh.

The purpose of this letter is to inform you that with no incremental effort, your application will also run on. the Lisa system. We will provide a Macintosh environment for the Lisa which allows Macintosh software to run standalone on the Lisa without any modification. Specifically, we will be marketing a single diskette which will boot the Lisa into a Macintosh environment and allow the Lisa to use the extentsive software base we expect to be offered for Macintosh.

From a user's perspective, using Macintosh software on the Lisa would work as follows:

- The user would boot the Lisa from a 3-1/2" microfloppy diskette using a microfloppy drive supplied for the Lisa.
- By then inserting their Macintosh application diskette, they are ready to work.

In addition to just being able to run Macintosh software, the user will also be able to take advantage of the additional memory in the Lisa as well as the larger screen. At some point in the future we also plan to have this environment support Lisa's hard disk.

So by following some simple rules in writing your Macintosh software (see attached), you will be able to leverage your efforts over both machines. We already have a substantial installed base of Lisa's which is growing daily. These Lisa users are anxious for the types of applications which you are developing for Macintosh and thus represent a sizable market to you.

I would strongly urge you to following the attached directions in writing your Macintosh applications. Not only will they insure your current ability to sell your software on the Lisa, but will also make it much more likely that your software will run on future Macintosh products. Additionally, I can provide you with information on copy protection implementations which will insure that your software is viable on both the Lisa as well as Macintosh.

If you have any questions, please either give me a call or don't hesitate to call Burt Cummings in the Lisa group. We are both here to help you succeed and are very excited about the prospects for the software you have underway.

Sincerely,

Mike Boich
Apple Computer, Inc.

Notes for applications concerned with LisaMac compatibility

Date: December 5,1983

The following is a compendium of suggestions intended to help guide anyone who wants to write software that will run on both MacIntosh, and Lisa 2.

(1) The size of the screen, or rowbytes, should never be assumed. An application can always determine the size of the screen by looking at the "bounds" field of the QuickDraw global variable "ScreenBits".

In Pascal this might look like:

```
thisScreenSize := ScreenBits.bounds
```

where thisScreenSize is of TYPE Rect.

In Assembly this might look like:

```
MOVE.L bounds(A0),(A0)        ; get start of screenbits.bounds
MOVE.L(A0)+,(A1)+             ; copy topLeft
MOVE.L(A0)+,(A1)+             ; copy bottomRight
```

where A0 is the address of the QuickDraw global "ScreenBits", and A1 points to our screen size.

(2) Use of sound should be limited to only the routine "SysBeep". Later on it may be possible to loosen this constraint to include access to the square wave generating capabilities of the sound driver.

(3) The size of memory should not be assumed. Memory size can be determined by using system routines such as "FreeMem".

(4) Most, if not all, attempts to access hardware directly (e.g. BTST #3,#$EXXXXX to see if the mouse button is up or down) will result in fatal system errors.

(5) In general, access to system globals should be through system routines. Perhaps later on there will be time to compile a list of those few global variables which, in fact, are not valid.

(6) Do not use the TAS (test and set) instruction of the 68000. A BSET instruction is not that much slower.

(7) The screen memory should not be accessed directly. All screen access should be through QuickDraw.

(8) The ROM code should not be accessed (i.e. jumped into) directly.

(9) The address of the dispatch table (used in replacing traps) should not be assumed. The address of individual traps can be determined by using the system routine

"GetTrapAddress".

(10) A program should not count on parameter memory being saved across system boots; "Time" will be saved however. In the case of a power loss, all parameter memory including Time will be initialized.

(11) Serial port "B" (i.e. one of the two serial ports) will not support 19.2k baud.

(12) Timing sensitive parts of programs should not be implemented with timing loops, or other application internal timing methods. Instead, use "ReadDateTime" for 1 second values or look at the "Ticks" global for ≈ 1/60 second values.

(13) Software protection????????

Date: March 21, 1984
To: Apple 32 Developers
From: Jeff Parrish
Re: Information update

Information update to developers:

Your programs need to be able to test whether they are running on a Macintosh or on a Lisa.

Byte 400009_H FF implies Lisa
Positive number implies Macintosh;

So, Tst.B $400009
BMI Lisastuff (label of your choice)
Beginning of Mac code

Date: 3 October 1983

To: Mac Developers

Re: Mac serial connector pinout

1 - GND
2 - +5 (may turn into an output handshake line-don't use!)
3 - GND
4 - TXD+
5 - TXD-
6 - +12 (for detecting power on ONLY!)
7 - HSK (CTS or TRxC, depending on 8530 mode)
8 - RXD+
9 - RXD-

For more-or-less RS232, use GND, TXD-, RXD-. The TXD+ and RXD+ signals provide RS422/423 compatibility.

The HSK (handshake) line (an input) connects to both CTS and to TRxC on the 8530. It can be used either for CTS, or for external clock depending on the mode the 8530 is in. As RS232 handshake, it usually connects to pin 20 on a DB-25.

For the Exceedingly Curious...

Signal lines go through a "deglitch" network, which is a "T" consisting of 25-50 ohm resistors and 200 pf to ground.

We use 26LS30 and 26LS32 interface chips between the 8530 SCC and the outside world.

The 8530 Data Carrier Detect lines (*DCDA pin 19 and *DCDB pin 21) are used as mouse inputs and generate interrupts used to detect mouse motion (!!!).

The Mouse Pinout

1 - GND
2 - +5 (Mouse ONLY!)
3 - GND
4 - X2 (to 6522 PB4)
5 - X1 (to 8530 *DCDA)
6 - No connect
7 - *SW (to 6522 PB3)
8 - Y2 (to 6522 PB5)
9 - Y1 (to 8530 *DCDB)

Bob Martin