Apple][Computer Family Technical Documentation

File Type Notes

Apple Computer -- Developer CD -- March 1993

```
##########################################################################
### FILE: FT.Assign.Form
##########################################################################
```

Request for File Type and Auxiliary Type Assignment
Apple Developer Technical Support

Before you ship your application, you must request file type and auxiliary type assignments for files you create from Apple Developer Technical Support. File type and auxiliary type combinations are used to identify files and their contents. The limited supply of file types requires that file types be defined generically and that specific auxiliary types be assigned by Apple for application use.

If you use a file type or auxiliary type which is not assigned to you, future versions of the Apple II or IIGS system software may identify your files with the wrong application, resulting in unpredictable results. For example, you might accidentally use a type for your data files that future system software will identify as code.

Apple assigns as many auxiliary types as you need in the appropriate file types. We also try to direct you towards file format standards that increase your application's ability to work with other applications. Assigned file type and auxiliary type combinations are used by the Finder(TM) to identify files on machines with sufficient memory.

For more information, please see "About File Type Notes" and the letter that comes with this form, or contact Developer Technical Support if you have any further questions.

If you are requesting more than one assignment, please copy this form and send us one copy for each assignment. You do not have to fill in the entire address if the forms are attached to each other. Thank you.

Send the completed form to:

Apple Developer Technical Support
Attn: Apple II File Type Assignment
20525 Mariani Ave. M/S 75-3T
Cupertino, CA 95014
AppleLink: AIIDTS
Internet: AIIDTS@AppleLink.Apple.com
MCI Mail: AIIDTS (264-0103)

Developer Name:_____

Address:_____

_____

Technical Contact:_____

Telephone (daytime):_____

Product Name (required):_____

Generic type of data in file:_____
(Use "About File Type Notes"

as a guide) _____

Kind of file as you wish it to
appear in the Finder(TM):_____
(30 characters maximum)
_____

END OF FILE FT.Assign.Form

```
##########################################################################
### FILE: FT.Letter
##########################################################################
```

Dear Apple II Developer:

Please find enclosed the form for a requested File Type and Auxiliary Type assignment for Apple II computers. Because the number of file types is severely limited (256 total), Apple Computer defines file types as generic descriptions of file contents and assigns specific auxiliary types to individual developers for specific file formats. These combinations must be assigned by Apple, not solely arbitrated, due to the specialized assignment nature of generic file type descriptions.

Please copy the form and use it to request all file type and auxiliary type assignments.

The following pointers may assist you and prevent delays when requesting an assignment:

- You may use previously documented file formats without obtaining an assignment for them (for example, you do not have to request an assignment for text files, Apple Preferred graphics files or Audio IFF sound files).

- You may use file types $F1 through $F8 at your convenience. Apple does not and will not arbitrate or document the use of these files.

- You should request an assignment if you create new file formats that you wish to identify from a directory entry. For example, text files are identified by a file type of $04. If you wish to be able to identify your files similarly, you should request an assignment.

- Many programs identify files only by file type and not by auxiliary type. These programs only identify your files by the general category of the file type, so it is important that we assign the best-suited type to your files. Please be as specific as possible in the "Generic type of data in file" field so we don't have to ask you for more information.

- All fields on the form are required and must not be left blank. If you cannot fill out a field (for example, your program doesn't have a final name yet), please use temporary file types and auxiliary types in file types $F1 through $F8 until all the information is available. If requesting multiple assignments, you may leave the address, technical contact and telephone number lines blank as long as the forms are attached to each other and one of them has all the fields completed.

- We can accommodate multiple requests (several combinations for one program), but we cannot accommodate requests where the auxiliary type field has been used as a field of flags. Only Apple Computer, Inc. can make such assignments, and then only for files that are system-wide and not specific to a particular application.

- When your application ships using your assignment, tell us. Otherwise we do not publish or acknowledge the assignment in any way to the world at large, and it is not included in the system software. We do not wish to pre-announce anyone's software.

- Please use a separate form for each requested file type and auxiliary type assignment.

If you can, we would appreciate the file format so we can publish it in an Apple II File Type Note. More information can be found in "About File Type Notes". If you have further questions, please feel free to contact us at the address on the registration form.

Thank you,

Apple Computer, Inc.
Developer Technical Support

END OF FILE FT.Letter

```
##########################################################################
### FILE: FTN.00.xxxx
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $00 (0)
Auxiliary Type: All

Full Name: Typeless file
Short Name: Unknown

Written by: Matt Deatherage March 1990

Files of this type and auxiliary type contain data that is unknown to the file's creator.

Files of type $00 contain data that is unknown to the program that creates the file. There are instances where programs, especially utilities, have to create files before they know the eventual file type and auxiliary type of the file. A good example of this is a telecommunications program that downloads a file without a Binary II or other header to preserve the file's attributes. Not knowing the file type, the program has little choice but to assign the file as an "unknown" type until such time as the real file type can be determined or assigned.

Files should be given type $00 when the creating program cannot determine the real file type. Reasonable guesses can be made (to continue the above example, a telecommunications program might assign file type $04 for all files transferred without protocol, guessing that ASCII transfers are probably for ASCII Text files).

File type $00 is not to be used for files regularly used by applications simply because the application programmer didn't wish to obtain a file type and auxiliary type assignment.

The auxiliary types for this file type are reserved; any files created of type $00 should be created with auxiliary type $0000.

```
### END OF FILE FTN.00.xxxx
```

```
##########################################################################
### FILE: FTN.01.xxxx
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $01 (1)
Auxiliary Type: All

Full Name: Bad Blocks file
Short Name: Bad blocks

Written by: Matt Deatherage **March 1990**

Files of this type and auxiliary type contain blocks that should not be used by the file system.

Files of type $01 contain physical media blocks that should not be used by the file system. A utility program can scan a disk looking for blocks that cannot be read and can then synthesize a file containing those blocks. Such a file should have type $01.

Files of this type should not be touched. If a program finds one, it should leave it alone. If you must delete such a file, be sure to double-check with the user.

Versions of ProDOS 8 prior to 1.8 can cause file system corruption (on ProDOS disks, of course) if you attempt to delete a file of bad blocks, so be sure not to do this. Attempting to delete a bad blocks file is only marginally reasonable for people manually attempting disk repair. A bad blocks scanning utility does not need to delete the old bad blocks files before creating new ones; it should just create new files as necessary. Apple Computer, Inc., does not support deleting a bad blocks file.

The auxiliary types for this file type are reserved; any files created of type $01 should be created with auxiliary type $0000.

```
### END OF FILE FTN.01.xxxx
```

```
##########################################################################
### FILE: FTN.08.0000
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $08
Auxiliary Type: $0000 to $3FFF

Full Name: Apple II Graphics File
Short Name: Graphics File

Revised by: Matt Deatherage May 1989
Written by: Matt Deatherage November 1988

Files of this type and auxiliary type contain standard Apple II graphics files.
Changes since November 1988: The offset was incorrectly listed as +121 instead of +120. The hexadecimal value of $78 is correct.

Files of type $08 and any auxiliary type less than or equal to $3FFF contain a standard Apple II graphics file in one of several modes. After determining that the auxiliary type is not $4000 or $4001 (which have been defined for high-resolution and double high-resolution pictures packed with the Apple IIGS PackBytes routine), you can determine the mode of the file by examining byte +120 (+$78). The value of this byte, which ranges from zero to seven, is interpreted as follows:

Mode	Page 1	Page 2
280 x 192 Black & White	0	4
280 x 192 Limited Color	1	5
560 x 192 Black & White	2	6
140 x 192 Full Color	3	7

Note that some modes only apply to high-resolution while some only apply to double high-resolution.

The format of the file is as follows:

+000 to +8191	Bytes	High-resolution image or portion of double high-resolution image stored in auxiliary memory.
+8192 to +16383	Bytes	Portion of double high-resolution image stored in main memory (not present for high-resolution).

File type $08 was originally defined as an Apple /// FotoFile, but now it is useful for those applications that wish to save high-resolution or double high-resolution data with a file type other than $06, which is a standard binary file. If you choose to use this type, you should remember that older applications which do not check the auxiliary type may attempt to interpret these files incorrectly.

END OF FILE FTN.08.0000

```
##########################################################################
### FILE: FTN.08.4000
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $08
Auxiliary Type: $4000

Full Name: Packed Apple II Hi-Res Graphics File
Short Name: Packed Hi-Res File

Written by: Matt Deatherage November 1988

Files of this type and auxiliary type contain a packed Apple II Hi-Res graphics screen.

Files of type $08 and auxiliary type $4000 contain a packed Apple II Hi-Res graphics screen which has been packed with the same algorithm that PackBytes on the Apple IIGS uses. This algorithm takes the 8K graphics screen and produces a file with an indeterminate length and internal format, so no "mode byte" at offset +121 is supported as it is with other files of type $08.

You can display a file of this type and auxiliary type by loading it, using UnPackBytes to decrypt the data, moving it into a high-resolution display buffer ($2000 or $4000 in the standard Apple II memory map), then simply toggling the appropriate display soft switches.

File type $08 was originally defined as an Apple /// FotoFile, but now it is useful for those applications that wish to save high-resolution or double high-resolution data with a file type other than $06, which is a standard binary file. If you choose to use this type, you should remember that older applications which do not check the auxiliary type may attempt to interpret these files incorrectly.

END OF FILE FTN.08.4000

```
##########################################################################
### FILE: FTN.08.4001
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $08
Auxiliary Type: $4001

Full Name: Packed Apple II Double Hi-Res Graphics File
Short Name: Packed Double Hi-Res File

Written by: Matt Deatherage November 1988

Files of this type and auxiliary type contain a packed Apple II Double Hi-Res graphics screen.

Files of type $08 and auxiliary type $4001 contain a packed Apple II Double Hi-Res graphics screen which has been packed with the same algorithm that PackBytes on the Apple IIGS uses. This algorithm takes the 16K graphics screen (auxiliary memory portion first) and produces a file with an indeterminate length and internal format, so no "mode byte" at offset +121 is supported as it is with other files of type $08.

You can display a file of this type and auxiliary type by loading it, using UnPackBytes to decrypt the data, moving the first half into auxiliary memory at location $2000, moving the second half into main memory at location $2000, then simply toggling the display soft switches and annunciators to turn on double high-resolution mode.

File type $08 was originally defined as an Apple /// FotoFile, but now it is useful for those applications that wish to save high-resolution or double high-resolution data with a file type other than $06, which is a standard binary file. If you choose to use this type, you should remember that older applications which do not check the auxiliary type may attempt to interpret these files incorrectly.

END OF FILE FTN.08.4001

```
##########################################################################
### FILE: FTN.19.xxxx
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $19 (25)
Auxiliary Type: All

Full Name: AppleWorks Data Base File
Short Name: AppleWorks DB File

Revised by: Matt Deatherage & John Kinder, Claris Corporation July 1990
Written by: Bob Lissner February 1984

Files of this type and auxiliary type contain an AppleWorks(R) Data Base file. Changes since September 1989: Corrected the description of offset +337 in the header.

Files of type $19 and any auxiliary type contain an AppleWorks Data Base file. AppleWorks is published by Claris. Claris also has additional information on AppleWorks files SEG.PR and SEG.ER. For information on AppleWorks, contact Claris at:

Claris Corporation
5201 Patrick Henry Drive
P.O. Box 58168
Santa Clara, CA 95052-8168

Technical Support
Telephone: (408) 727-9054
AppleLink: Claris.Tech

Customer Relations
Telephone: (408) 727-8227
AppleLink: Claris.CR

AppleWorks was created by Bob Lissner. AppleWorks 2.1 was done by Bob Lissner and John Kinder of Claris. AppleWorks 3.0 was done by Randy Brandt, Alan Bird and Rob Renstrom of Beagle Bros Software with John Kinder of Claris.

Definitions

The following definitions apply to AppleWorks files in addition to those defined for all Apple II file types:

MRL	Data base multiple record layout
SRL	Data base single record layout
RAC	Review/Add/Change screen
DB	AppleWorks or /// E-Z Pieces Data Base
SS	AppleWorks or /// E-Z Pieces Spreadsheet
WP	AppleWorks or /// E-Z Pieces Word Processor

AW AppleWorks or /// E-Z Pieces

Auxiliary Type Definitions

The volume or subdirectory auxiliary type word for this file type is defined to control uppercase and lowercase display of filenames. The highest bit of the least significant byte corresponds to the first character of the filename, the next highest bit of the least significant byte corresponds to the second character, etc., through the second bit of the most significant byte, which corresponds to the fifteenth character of the filename.

AppleWorks performs the following steps when it saves a file to disk:

1. Zeros all 16 bits of the auxiliary type word.
2. Examines the filename for lowercase letters. If one is found, it changes the corresponding bit in the auxiliary type word to 1 and changes the letter to uppercase.
3. Examines the filename for spaces. If one is found, it changes the corresponding bit in the auxiliary type word to 1 and changes the space to a period.

When files are read from disk, the filename and auxiliary type information from the directory file entry are used to determine which characters should be lowercase and which periods should be displayed as spaces. If you use the auxiliary type bytes for a different purpose, AppleWorks will still display the filenames, but the wrong letters are likely lowercase.

File Version Changes

Certain features present in AppleWorks 3.0 files are not backward-compatible to 2.1 and earlier versions. Such features are noted in the text. AppleWorks Data Base files which may not be loaded by versions prior to 3.0 are identified by a non-zero byte at location +218, referred to as location DBMinVers.

Those features added for AppleWorks 2.0, 2.1 and 3.0 not previously documented are indicated with that version number in the margin.

Data Base Files

Data base files start with a variable length header, followed by 600 bytes for each report format (if any), the standard values record, then variable length information for each record.

Header Record

The header contains category names, record selection rules, counts, screen positioning information, and all other non-record specific information.

+000 to +001	Word	The number of bytes in the remainder of the header record. Use this count for your next ProDOS read from the disk.
+002 to +029		Ignore these bytes.
+030	Byte	Cursor direction when the Return key

			is pressed in SRL. $01: Order in which you defined categories or $02: Left to right, top to bottom.
	+031	Byte	What direction should the cursor go when you press the Return key in the MRL? D)own or R)ight.
	+032 to +033		Ignore these bytes.
	+034	Byte	Style of display that Review/Add/Change was using when the file was saved: R: SRL. Slash (/): MRL.
	+035	Byte	Number of categories per record. Values from $01 to $1E.
3.0	+036 to +037	Word	Number of records in file. If DBMinVers is non-zero, the high bit of this word may be set. If it is, there are more than eight reports and the remaining 15 bits contain the true number of records defined.
3.0	+038	Byte	Number of reports in a file, maximum of 8 (20 for 3.0).
	+039 to +041		Ignore these bytes.
	+042 to +071	Bytes	For each of up to 30 columns, showing the number of spaces used for this column on the MRL. Be sure you understand that categories may have been rearranged on the MRL. Byte +042 refers to the leftmost column on the MRL.
	+072 to +077		Ignore six bytes.
	+078 to +107	Bytes	For up to 30 categories on the MRL, the defined category that appears in each position. Byte +078 is the leftmost column of the MRL and has a value from $01 to $1E that defines which of the category names appears in this position. These numbers change as a result of changing the layout of the MRL.
	+108 to +113		Ignore six bytes.
	+114 to +143	Bytes	For up to 30 categories on the SRL, the horizontal screen position. These are changed as a result of changing the layout of the SRL. AppleWorks makes sure that these entries, and the vertical screen positions, are kept in order from left to right within top to bottom.
	+144 to +149		Ignore these six bytes.
	+150 to +179	Bytes	For up to 30 categories on the SRL, the vertical screen position.
	+180 to +185		Ignore six bytes.
	+186 to +215	Bytes	For up to 30 categories on the SRL, which of the category names appears in this position. These change as a result of changing the SRL. This number refers to the category names listed below.

	+216 to +217		Ignore two bytes.
3.0	+218	Byte	DBMinVers. The minimum version of AppleWorks needed to read this file. This will be $00 unless there are more than 8 report formats; it will then contain the version number 30 ($1E) or greater.
3.0	+219	Byte	The first frozen column in the titles.
3.0	+220	Byte	If this is zero, no titles are present. If non-zero, this is the last frozen column.
3.0	+221	Byte	Leftmost active column. This is zero-based; if this value is zero, it means column one, etc.
	+222	Byte	Number of categories on MRL. Will be less than or equal to the number of categories in the file. SRL displays all categories, so there is no equivalent number for SRL.
	+223 to +224	Word	For the first line of RAC selection rules. Zero means no selection rules, while any other value refers to the category name that is tested. The high byte will always be zero.
	+225 to +226	Word	Category name for the second line of RAC selection rules. Zero means that there is only one line.
	+227 to +228	Word	Category name for the third line of RAC selection rules. Zero means that there is no third line.
	+229 to +230	Word	For the first line of RAC rules, which of the tests is to be applied. 1 means equals, 2 means greater than and so on.
	+231 to +232	Word	Test for the second line of rules, if any.
	+233 to +234	Word	Test for the third line, if any.
	+235 to +236	Word	Continuation code for the first line: 1: And, 2: Or, 3: Through.
	+237 to +238	Word	Continuation code for the second line.
	+239 to +240	Word	Continuation code for the third line. Not possible, so it is always zero.
	+241 to +272	String	Maximum length of 30 bytes. Comparison information for the first line RAC selection rules.
	+273 to +304	String	Comparison for the second line.
	+305 to +336	String	Comparison for the third line.
	+337 to +356		Ignore these twenty bytes.
	+357 to +378	String	Name of the first category. Maximum length of 20 bytes. If the file has only one category, the header record will end here.
	+379 to +400	String	Name of the second category, if any. This area will not be on the header record if there is only one category.
	+401	22 Bytes	Additional 22 byte entries for all remaining categories. The size of the header record depends on the

		number of categories. Space is not maintained past the last category.

Report Records

Report records follow the header record. One of the header record categories tells you how many report records to expect. The number will be from zero to eight. Each report record is 600 bytes, and contains:

+000 to +019	String	Report name. Maximum length of 19 characters.
+020 to +052	Bytes	Column width for up to 33 columns in a tables-style report format. Byte +020 is for the leftmost column on a tables-style report. There can be up to 30 categories from the file, plus 3 more calculated columns. For labels-style report formats, the value is a byte that has the horizontal position of this category, relative to the left margin.
+053 to +055		Skip 3 bytes.
+056 to +088	Bytes	For tables-style: Number of spaces to be printed at the right of justified columns. For labels-style: Vertical position on the report for each of up to 30 categories. A value of 1 means that category is on the first line of labels-style report.
+089 to +091		Skip 3 bytes.
+092 to +124	Bytes	For up to 33 columns of tables-style: Values from 1 to 30 refer to which category name appears in this column on the report. Values of $80, $81 and $82 are the three calculated categories, from left to right. For labels-style: Same as tables-style, minus the calculated categories.
+125 to +127		Skip these three bytes.
+128 to +160	Bytes	For up to 33 columns of tables-style: $99 means no foot totals, 0 through 4 means the number of decimal places for a foot total. For labels-style: For up to 30 categories on report, Boolean bytes whether or not category names are to be printed.
+161 to +163		Skip these three bytes.
+164 to +196	Bytes	For up to 33 columns of tables-style: $99 means left justified, 0 through 4 means right justified with 0 to 4 decimal places. For up to 30 categories of labels-style: Boolean bytes whether or not to float (OA-J) this category up against the category to its left.

+197 to +199		Skip three bytes.
+200	Byte	Number of categories on report. Includes calculated categories, if any.
+201	Byte	Tables-style. If there is at least one calculated category, this contains values from 1 to 33: which column of the report. Labels-style: Values from 3 to 21. Position of the line on the screen that says "Each record will print nn lines."
+202	Byte	Tables-style: Same as +201, but for the second calculated category, if any. Labels-style: Unused.
+203	Byte	Tables-style: Same as +201, but for the third calculated category, if any. Labels-style: Unused
+204	Byte	Tables-style only: If there is a group total column, this byte states which of the category names is used as a basis. Values from 1 to 30.
+205	Byte	Platen width value, in 10ths of an inch. For example, a value of 8.0 inches entered by the user will show as 80 or $50.
+206	Byte	Left margin value. All inches values are in 10ths.
+207	Byte	Right margin value.
+208	Byte	Characters per inch.
+209	Byte	Paper length value, in 10ths of an inch.
+210	Byte	Top margin value.
+211	Byte	Bottom margin value.
+212	Byte	Lines per inch. 6 or 8.
+213	Byte	Not relevant. Probably always a "C."
+214	Byte	Type of report format. H: tables-style, V: labels-style.
+215	Byte	Spacing: S(ingle, D(ouble, or T(riple. Expect these three letters, even in European versions.
+216	Byte	Print report header. Boolean.
+217	Byte	Tables-style: If user has specified group totals, Boolean, just print the group totals.
+218	Byte	Labels-style: Boolean, omit the line when all entries on the line are blank.
+219	Byte	Labels-style: Boolean, keep the number of lines the same within each record.
+220 to +301	String	80-byte string. Title line, if any.
+302 to +323	String	Tables-style. 20-byte string. Name of the first calculated category, if any.
+324 to +355	String	Tables-style. 30-byte string. Calculation rules for first calculated category, if any.
+356 to +409	String	Tables-style. Name and rules for second calculated category, if any.
+410 to +463	String	Tables-style. Name and rules for third calculated category, if any.
+464 to +477	String	If user has specified "Send special

		codes to printer," this is a 13 byte string containing those codes.
+478	Byte	Boolean: Print a dash when an entry is blank.
+479 to +592	Words & Strings	Record selection rules. Exact same format as described in the header record.
+593 to +599		Unused

Data Records

Data records follow the report records. The first data record contains the standard values. Each following data record corresponds to one data base record.

These records contain all of the categories within one stream of data. The category entries are in the same order that the category names appear in the header record.

Bytes +0 and +1 are a word that contains a count of the number of bytes in the remainder of the record.

Byte +2 of each record will always be a control byte. Other control bytes within each record define the contents of the record. Control bytes may be:

$01-$7F	This is a count of the number of following bytes that are the contents of a category.
$81-$9E	This (minus $80) is a count of the number of categories to be skipped. For example, $82 means skip two categories.
$FF	This indicates the end of the record.

The information in individual categories may have some special coding so that date and time entries can be arranged (sorted).

Date entries have the following format:

+000	Byte	$C0 (192). Identifies a date entry.
+001 to 002	Two bytes	ASCII year code, like "84" ($38 $34).
+003	Byte	ASCII month code. A means January, L means December.
+004 to +005	Two bytes	ASCII day of the month, like "31" ($33 $31).

Time entries have the following format:

+000	Byte	$D4 (212). Identifies a time entry.
+001	Byte	ASCII hour code. A means 00 (the hour after midnight). X means 23, the hour before midnight.
+002 to +003	Two bytes	ASCII minute code. Values from 00 to 59.

File Tags

All AppleWorks files normally end with two bytes of $FF; tags are

anything after that. Although File Tags were primarily designed by Beagle Bros, they can be used by any application that needs to create or modify an AppleWorks 3.0 file.

Because versions of AppleWorks before 3.0 stop at the double $FF, they simply ignore tags.

The File Tag structure is as follows:

+000	Byte	Tag ID. Should be $FF.
+001	Byte	2nd ID byte. These values will be defined and arbitrated by Beagle Bros Software. Beagle may be reached at: Beagle Bros Inc 6215 Ferris Square, #100 San Diego, CA 92121
+002 to +003	Word	Data length. If this is the last tag on the file, the low byte (+002) will be a count of the tags in this file, and the high byte (+003) will be $FF.
+004 to nnn	Bytes	Actual tag data, immediately followed by the next four-byte tag ID. These bytes do not exist for the last tag.

There is a maximum of 64 tags per file. Each tag may be no larger than 2K.

END OF FILE FTN.19.xxxx

```
##########################################################################
### FILE: FTN.1A.xxxx
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: **$1A (26)**
Auxiliary Type: **All**

Full Name: **AppleWorks Word Processor File**
Short Name: **AppleWorks WP File**

Revised by: **Matt Deatherage & John Kinder, CLARIS Corp. September 1989**
Written by: **Bob Lissner February 1984**

Files of this type and auxiliary type contain an AppleWorks(R) Word Processor file.
Changes since May 1989: Updated to include AppleWorks 2.1 and AppleWorks 3.0.

Files of type $1A and any auxiliary type contain an AppleWorks Word Processor file. AppleWorks is published by CLARIS. CLARIS also has additional information on AppleWorks files SEG.PR and SEG.ER. For information on AppleWorks, contact CLARIS at:

CLARIS Corporation
5201 Patrick Henry Drive
P.O. Box 58168
Santa Clara, CA 95052-8168

Technical Support
Telephone: (408) 727-9054
AppleLink: Claris.Tech

Customer Relations
Telephone: (408) 727-8227
AppleLink: Claris.CR

AppleWorks was created by Bob Lissner. AppleWorks 2.1 was done by Bob Lissner and John Kinder of CLARIS. AppleWorks 3.0 was done by Alan Bird, Rob Renstrom and Randy Brandt of Beagle Bros Software with John Kinder of CLARIS.

Definitions

The following definitions apply to AppleWorks files in addition to those defined for all Apple II file types:

MRL	**Data base multiple record layout**
SRL	**Data base single record layout**
RAC	**Review/Add/Change screen**
DB	**AppleWorks or /// E-Z Pieces Data Base**

SS	AppleWorks or /// E-Z Pieces Spreadsheet
WP	AppleWorks or /// E-Z Pieces Word Processor
AW	AppleWorks or /// E-Z Pieces

Auxiliary Type Definitions

The volume or subdirectory auxiliary type word for this file type is defined to control uppercase and lowercase display of filenames. The highest bit of the least significant byte corresponds to the first character of the filename, the next highest bit of the least significant byte corresponds to the second character, etc., through the second bit of the most significant byte, which corresponds to the fifteenth character of the filename.

AppleWorks performs the following steps when it saves a file to disk:

1. Zeros all 16 bits of the auxiliary type word.
2. Examines the filename for lowercase letters. If one is found, it changes the corresponding bit in the auxiliary type word to 1 and changes the letter to uppercase.
3. Examines the filename for spaces. If one is found, it changes the corresponding bit in the auxiliary type word to 1 and changes the space to a period.

When files are read from disk, the filename and auxiliary type information from the directory file entry are used to determine which characters should be lowercase and which periods should be displayed as spaces. If you use the auxiliary type bytes for a different purpose, AppleWorks will still display the filenames, but the wrong letters are likely lowercase.

File Version Changes

Certain features present in AppleWorks 3.0 files are not backward-compatible to 2.1 and earlier versions. Such features are noted in the text. AppleWorks Word Processor files which may not be loaded by versions prior to 3.0 are identified by a non-zero byte at location +183, referred to as location SFMinVers.

Those features added for AppleWorks 2.0, 2.1 and 3.0 not previously documented are indicated with that version number in the margin.

Word Processor Files

Word Processor files start with a 300 byte header, followed by a number of variable length line records, one for each line on the screen.

Header Record

The header contains the following information:

+000 to +003		Not used.
+004	Byte	$4F (79)
+005 to +084	Bytes	Tab stops. Either equal sign (=) or vertical bar (|) If SFMinVers is non-zero, these will be one of the following values:

		"=" - no tab "<" - left tab "^" - center tab ">" - right tab "." - decimal tab.
+085	Byte	Boolean: Zoom switch.
+086 to +089		Four bytes not used.
+090	Byte	Boolean: Whether file is currently paginated (i.e., whether the page break lines are displayed).
+091	Byte	Minimum left margin that should be added to the margin that is appearing on the screen. This is normally one inch, shown in 10ths of an inch, 10 or $0A.
+092	Byte	Boolean: Whether file contains any mail-merge commands.
+093 to +175	Bytes	Not used. Reserved.
3.0 +176	Byte	Boolean: Whether there are multiple rulers in the document.
3.0 +177 to +182	Bytes	Used internally for keeping track of tab rulers.
3.0 +183	Byte	SFMinVers. The minimum version of AppleWorks needed to read this document. If this document contains 3.0 specific features (tabs and multiple tab rulers, for example), this byte will contain the version number 30 ($1E). Otherwise, it will be zero ($00).
+184 to +249	Bytes	Reserved.
+250 to +299	Bytes	Available. Will never be used by AppleWorks. If you are creating this type of file, you can use this area to keep information that is important to your program.

Line Records

Line records are of three different types. The first line record after the 300 byte header corresponds to line 1, the next is line 2, and so on. The first two bytes of each line record contain enough information to establish the type.

If SFMinVers is non-zero, the first line record (two bytes long) is invalid and should be skipped.

Carriage Return Line Records

Carriage return line records have a $D0 (208) in byte +001. Byte +000 is a one byte integer between 00 and 79 that is the horizontal screen position of this carriage return.

Command Line Records

Command line records are formatting commands that appear on the screen in the form:

--------Double Space

for example. These records can be identified by a value greater than $D0 (208) in byte +001. They are:

	Byte +001	Command	Byte +000	
3.0	$D4	reserved	(used internally as ruler)	
3.0	$D5	Page header end		
3.0	$D6	Page footer end		
3.0	$D7	Right justified		
	$D8	Platen width	Byte	10ths of an inch
	$D9	Left margin	Byte	10ths of an inch
	$DA	Right margin	Byte	10ths of an inch
	$DB	Chars per inch	Byte	
	$DC	Proportional-1		No meaning
	$DD	Proportional-2		
	$DE	Indent	Byte	Characters
	$DF	Justify		
	$E0	Unjustify		
	$E1	Center		
	$E2	Paper length	Byte	10ths of an inch
	$E3	Top margin	Byte	10ths of an inch
	$E4	Bottom margin	Byte	10ths of an inch
	$E5	Lines per inch	Byte	
	$E6	Single space		
	$E7	Double space		
	$E8	Triple space		
	$E9	New page		
	$EA	Group begin		
	$EB	Group end		
	$EC	Page header		
	$ED	Page footer		
	$EE	Skip lines	Byte	Count
	$EF	Page number	Byte	
	$F0	Pause each page		
	$F1	Pause here		
	$F2	Set marker	Byte	Marker number
	$F3	Page number	Byte	(add 256)
	$F4	Page break	Byte	Page number
	$F5	Page break	Byte	(add 256)
	$F6	Page break	Byte	(break in middle of paragraph)
	$F7	Page break	Byte	(add 256 in middle of paragraph)
	$FF	End of file		

Text Records

Text records are the lines where text has been typed. The format is:

+000 to +001	Word	Number of bytes following this word. Since the maximum is about 80, byte +001 is always zero. Use byte +001 to identify text lines.
3.0 +002		If bit 7 is on, this line contains Tab and Tab Filler special codes (described below). The remaining seven bits are the screen column for the first text character. Usually will be zero, but may

vary as a result of left margin, centering, and indent commands. If this byte is $FF, this text line is actually a ruler--ASCII equivalent of what appears on the top of the screen.

+003 Byte — If bit 7 (the high bit) of this byte is on, there is a carriage return on the end of this line. If off, no carriage return. Bits 6-0: Number of bytes of text following this byte.

+004 to nnn — Actual text bytes. Consists of ASCII characters and special codes. The special codes are values from $01 to $1F, and indicate special formatting features:

	Code	Meaning
	$01	Begin boldface
	$02	Boldface end
	$03	Superscript begin
	$04	Superscript end
	$05	Subscript begin
	$06	Subscript end
	$07	Underline begin
	$08	Underline end
	$09	Print page number
	$0A	Enter keyboard
	$0B	Sticky space
	$0C	Begin Mail merge
3.0	$0D	Reserved
3.0	$0E	Print Date
3.0	$0F	Print Time
3.0	$10	Special Code 1
3.0	$11	Special Code 2
3.0	$12	Special Code 3
3.0	$13	Special Code 4
3.0	$14	Special Code 5
3.0	$15	Special Code 6
3.0	$16	Tab character
3.0	$17	Tab fill character (used in formatting lines)
3.0	$18	Reserved

File Tags

All AppleWorks files normally end with two bytes of $FF; tags are anything after that. Although File Tags were primarily designed by Beagle Bros, they can be used by any application that needs to create or modify an AppleWorks 3.0 file.

Because versions of AppleWorks before 3.0 stop at the double $FF, they simply ignore tags.

The File Tag structure is as follows:

+000 Byte — Tag ID. Should be $FF.

+001	Byte	2nd ID byte. These values will be defined and arbitrated by Beagle Bros Software. Beagle may be reached at: Beagle Bros Inc 6215 Ferris Square, #100 San Diego, CA 92121
+002 to +003	Word	Data length. If this is the last tag on the file, the low byte (+002) will be a count of the tags in this file, and the high byte (+003) will be $FF.
+004 to nnn	Bytes	Actual tag data, immediately followed by the next four-byte tag ID. These bytes do not exist for the last tag.

There is a maximum of 64 tags per file. Each tag may be no larger than 2K.

END OF FILE FTN.1A.xxxx

```
########################################################################
### FILE: FTN.1B.xxxx
########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $1B (27)
Auxiliary Type: All

Full Name: AppleWorks Spreadsheet File
Short Name: AppleWorks SS File

Revised by: Matt Deatherage & John Kinder, CLARIS Corp. September 1989
Written by: Bob Lissner February 1984

Files of this type and auxiliary type contain an AppleWorks(R) Spreadsheet file.
Changes since May 1989: Updated to include AppleWorks 2.1 and AppleWorks 3.0.

Files of type $1B and any auxiliary type contain an AppleWorks Spreadsheet file. AppleWorks is published by CLARIS. CLARIS also has additional information on AppleWorks files SEG.PR and SEG.ER. For information on AppleWorks, contact CLARIS at:

CLARIS Corporation
5201 Patrick Henry Drive
P.O. Box 58168
Santa Clara, CA 95052-8168

Technical Support
Telephone: (408) 727-9054
AppleLink: Claris.Tech

Customer Relations
Telephone: (408) 727-8227
AppleLink: Claris.CR

AppleWorks was created by Bob Lissner. AppleWorks 2.1 was done by Bob Lissner and John Kinder of CLARIS. AppleWorks 3.0 was done by Rob Renstrom, Randy Brandt and Alan Bird of Beagle Bros Software with John Kinder of CLARIS.

Definitions

The following definitions apply to AppleWorks files in addition to those defined for all Apple II file types:

MRL	Data base multiple record layout
SRL	Data base single record layout
RAC	Review/Add/Change screen
DB	AppleWorks or /// E-Z Pieces Data Base

SS	AppleWorks or /// E-Z Pieces Spreadsheet
WP	AppleWorks or /// E-Z Pieces Word Processor
AW	AppleWorks or /// E-Z Pieces

Auxiliary Type Definitions

The volume or subdirectory auxiliary type word for this file type is defined to control uppercase and lowercase display of filenames. The highest bit of the least significant byte corresponds to the first character of the filename, the next highest bit of the least significant byte corresponds to the second character, etc., through the second bit of the most significant byte, which corresponds to the fifteenth character of the filename.

AppleWorks performs the following steps when it saves a file to disk:

1. Zeros all 16 bits of the auxiliary type word.
2. Examines the filename for lowercase letters. If one is found, it changes the corresponding bit in the auxiliary type word to 1 and changes the letter to uppercase.
3. Examines the filename for spaces. If one is found, it changes the corresponding bit in the auxiliary type word to 1 and changes the space to a period.

When files are read from disk, the filename and auxiliary type information from the directory file entry are used to determine which characters should be lowercase and which periods should be displayed as spaces. If you use the auxiliary type bytes for a different purpose, AppleWorks will still display the filenames, but the wrong letters are likely lowercase.

File Version Changes

Certain features present in AppleWorks 3.0 files are not backward-compatible to 2.1 and earlier versions. Such features are noted in the text. AppleWorks spreadsheet files which may not be loaded by versions prior to 3.0 are identified by a non-zero byte at location +242, referred to as location SSMinVers.

Those features added for AppleWorks 2.0, 2.1 and 3.0 not previously documented are indicated with that version number in the margin.

Spreadsheet Files

Spreadsheet files start with a 300 byte header record that contains basic information about the file, including column widths, printer options, window definitions, and standard values.

Header Record

The spreadsheet header record contains the following entries:

+000 to +003		Skip 4 bytes.
+004 to +130	Bytes	The column width for each column.
+131	Byte	Order of recalculation. ASCII R or C.
+132	Byte	Frequency of recalculation. ASCII A or M.

+133 to +134	Word	Last row referenced.
+135	Byte	Last column referenced.
+136	Byte	Number of windows: ASCII 1: just one window, S: side by side windows, T: top and bottom windows.
+137	Byte	Boolean: If there are two windows, are they synchronized?
+138 to +161		The next 20 (approximately) variables are for the current window. If there is only one window, it is the current window. If there are two windows, the current window is the window that had the cursor in it.
+138	Byte	Window standard format for label cells. 2: left justified, 3: right justified, 4: centered.
+139	Byte	Window standard format for value cells. 2: fixed, 3: dollars, 4: commas, 5: percent, 6: appropriate
+140	Byte	More of window standard format for value cells. Number of decimal places to display. Values from 0 to 7.
+141	Byte	Top screen line used by this window. This is the line that the =====A=========B==== appears on. Normally 1 unless there are top and bottom windows.
+142	Byte	Leftmost screen column used by this window. This is the column that the hundreds digit of the row number appears in. Normally 0 unless there are side-by-side windows.
+143 to +144	Word	Top, or first, row appearing in titles area. This will probably be 0 if there are no top titles.
+145	Byte	Leftmost, or first, column appearing in left titles area. This will probably be 0 if there are no left titles.
+146 to +147	Word	Last row appearing in top titles area. This will probably be zero if there are no top titles.
+148	Byte	Last column appearing in left titles area. This will probably be zero if there are no left titles.
+149 to +150	Word	Top, or first, row appearing in the body of the window. The body is defined as those rows that are on the screen, but not in the titles area.
+151	Byte	Leftmost, or first, column appearing in the body of the window.
+152	Byte	The screen line that the top body row goes on. Normally 2, unless there are top titles or top and bottom windows.
+153	Byte	Leftmost screen column used for the

		leftmost body column. Normally 4 unless there are side titles, or side-by-side windows.
+154 to +155	Word	Bottom, or last, row appearing in this window.
+156	Byte	Rightmost, or last, column appearing in this window.
+157	Byte	The screen line that the last body row goes on. Normally $13 (19) unless there are top and bottom windows.
+158	Byte	The rightmost screen column used by this window. Normally $4E (78) unless there are side-by-side windows.
+159	Byte	Number of horizontal screen locations used to display the body columns. Normally $48 (72), because 8 columns of 9 characters each are the standard display. This is affected by side-by-side windows, side titles, and variable column widths.
+160	Byte	Boolean: Rightmost column is not fully displayed. This can only happen when the body portion of the window is narrower than the width of a particular column.
+161	Flag Byte	Titles switch for this window. Bit 7: top titles, Bit 6: side titles. These bits represent top titles, side titles, both, and no titles.
+162 to +185		Window information for the second window. This is meaningful only if there are two windows. This is the information for the window that the cursor is not currently in. See the descriptions for the current window (+138 to +161).
+186 to +212		Not currently used.
+213	Byte	Boolean: Cell protection is on or off.
+214		Not currently used.
+215	Byte	Platen width value, in 10ths of an inch. For example, a value of 80 inches entered by the user will show as 80 or $50.
+216	Byte	Left margin value. All inches values are in 10ths of an inch.
+217	Byte	Right margin value.
+218	Byte	Characters per inch.
+219	Byte	Paper length value, in 10ths of an inch.
+220	Byte	Top margin value.
+221	Byte	Bottom margin value.
+222	Byte	Lines per inch. 6 or 8.
+223	Byte	Spacing: S(ingle, D(ouble, or T(riple. Expect these three letters, even in European versions.

+224 to +237	Bytes	If user has specified "Send special codes to printer," this is a 13-byte string containing those codes.
+238	Byte	Boolean: Print a dash when an entry is blank.
+239	Byte	Boolean: Print report header.
+240	Byte	Boolean: Zoomed to show formulas.
2.1 +241	Byte	Reserved; used internally.
3.0 +242	Byte	SSMinVers. The minimum version of AppleWorks needed to read this document. If this document contains version 3.0-specific functions (such as calculated labels or new functions), this byte will contain the version number 30 ($1E). Otherwise, it will be zero ($00).
+243 to +249		Reserved for future use.
+250 to +299		Available. Will never be used by AppleWorks. If you are creating these files, you can use this area to keep information that is important to your program.

Row Records

Row records contain a variable amount of information about each row that is non-blank. Each row record contains enough information to completely build one row of the spreadsheet:

3.0 +000 to +001	Word	Number of additional bytes to read from disk. $FFFF means end of file. If SFMinVers is not zero, these two bytes are invalid and should be skipped. The first row record begins at +302 in an AW 3.0 SS file.
+002 to +003	Word	Row number.
+004	Byte	Beginning of actual information for the row. This byte of each record will always be a control byte. Other control bytes within each record define the contents of the record. Control bytes may be:

$01-$7F	This is a count of the number of following bytes that are the contents of a cell entry.
$81-$FE	This (minus $80) is a count of the number of columns to be skipped. For example, $82 means skip two columns.
$FF	This indicates the end of the row.

Cell Entries

Cell entries contain all the information that is necessary to build one cell. There are several types:

Value Constants

Value constants are cells that have a value that cannot change. This means that someone typed a constant into the cell, 3.14159, for example.

+000	Flag Byte	Bit 7 is always on. Bit 6 on means that if the value is zero, display a blank instead of a zero. This is for pre-formatted cells that still have no value. Bit 5 is always on. Bit 4 on means that labels cannot be typed into this cell. Bit 3 on means that values cannot be typed into this cell. Bits 2,1, and 0 specify the formatting for this cell: 1 Use spreadsheet standard 2 Fixed 3 Dollars 4 Commas 5 Percent 6 Appropriate
+001	Flag Byte	Bit 7 is always zero. Bit 6 is always zero. Bit 5 is always zero. Bit 4 on indicates that this cell must be calculated the next time this spreadsheet is calculated, even if none of the referenced cells are changed. This bit makes sense on for cells that have a alculated formula. Bits 2, 1, and 0: Number of decimal places for fixed, dollars, commas, or percent formats.
+002 to +009		8-byte SANE double format floating point number.

Value Labels

Note: The entire Value Labels cell record entry requires AppleWorks 3.0 or later.

Value labels are cells whose function has returned a label value. Formulas like @Lookup, @Choose and @IF can all return labels as their results. Specific format:

+000	Flag Byte	Bit 7 is always one. Bit 6 on means not to display the cell. This was originally intended for pre-formatted cells that still have no value. If a value is placed in this cell, be sure to turn this bit off.

		Bit 5 is always zero. Bits 4, 3, 2, 1, and 0 are the same as regular label cells.
+001	Flag Byte	Bit 7 is always one. Bit 6 set indicates the last evaluation of this formula resulted in @NA. Bit 5 set indicates the last evaluation of his formula resulted in @Error. Bit 4 on indicates that this cell must be calculated the next time this spreadsheet is calculated, even if none of the referenced cells are changed. Bit 3 is always one. Bits 2 - 0 are ignored.
+002 to nnn	String	Pascal string containing characters to display.
+nnn+1 to xxx	Bytes	Various control bytes that are "tokens" representing the formula that was typed by the user. They are defined below.

Value Formulas

Value formulas are cells that contain information that has to be evaluated. Formulas like AA17+@sum(r19...r21) and @Error are examples. Specific format:

+000	Flag Byte	Bit 7 is always on. Bit 6 on means to not display the cell. This was originally intended for pre-formatted cells that still have no value. If a value is placed in this cell, be sure to turn off this bit. Bit 5 is always off. Bits 4, 3, 2, 1, and 0 are the same as value constants.
+001		Bit 7 is always on. Bit 6 on indicates that the last evaluation of this formula resulted in an @NA. Bit 5 on indicates that the last evaluation of this formula resulted in an @Error. Bits 4, 2, 1, and 0 are the same as value constants.
+002 to +009		8-byte SANE double floating point number that is the most recent evaluation of this cell.
+010 to nnn		Various control bytes that are tokens representing the formula that was entered by the user. They are:

Byte	Means
3.0	$C0 @Deg
3.0	$C1 @Rad
3.0	$C2 @Pi
3.0	$C3 @True
3.0	$C4 @False
3.0	$C5 @Not

3.0	$C6	@IsBlank
3.0	$C7	@IsNA
3.0	$C8	@IsError
3.0	$C9	@Exp
3.0	$CA	@Ln
3.0	$CB	@Log
3.0	$CC	@Cos
3.0	$CD	@Sin
3.0	$CE	@Tan
3.0	$CF	@ACos
3.0	$D0	@ASin
3.0	$D1	@ATan2
3.0	$D2	@ATan
3.0	$D3	@Mod
3.0	$D4	@FV
3.0	$D5	@PV
3.0	$D6	@PMT
3.0	$D7	@Term
3.0	$D8	@Rate
2.0	$D9	@Round
2.0	$DA	@Or
2.0	$DB	@And
	$DC	@Sum
	$DD	@Avg
	$DE	@Choose
	$DF	@Count
	$E0	@Error (followed by 3 bytes of zero)
3.0	$E1	@IRR
	$E2	@If
	$E3	@Int
	$E4	@Lookup
	$E5	@Max
	$E6	@Min
	$E7	@NA (followed by three bytes of zero)
	$E8	@NPV
	$E9	@Sqrt
	$EA	@Abs
	$EB	Not currently used
	$EC	Not equal (<>)
	$ED	greater than or equal to (>=)
	$EE	less than or equal to (<=)
	$EF	equals (=)
	$F0	greater than (>)
	$F1	less than (<)
	$F2	comma (,)
	$F3	exponentiation sign (^)
	$F4	right parenthesis (")")
	$F5	minus (-)
	$F6	plus (+)
	$F7	divide (/)
	$F8	multiply (*)
	$F9	left parenthesis ("(")
	$FA	unary minus (-) i.e., -A3
	$FB	(unary plus (+) i.e., +A3)
	$FC	ellipses (...)
	$FD	Next 8 bytes are SANE

			double number
		$FE	Next 3 bytes are row, column reference
	3.0	$FF	Next n bytes are a Pascal string

Three of the codes require special information. Code $FD indicates that the next 8 bytes are a SANE numerics package double precision floating point number. All constants within formulas are carried in this manner.

Code $FE indicates that the next three bytes point at a cell:

+000	Byte	$FE
+001	Byte	Column reference. Add this byte to the column number of the current cell to get the column number of the pointed at cell. This value is sometimes negative, but Add always works.
+002 to +003	Word	Row reference. Add this word to the row number of the current cell to get the row number of the pointed at cell. This value is sometimes negative, but Add always works.

Code $FF indicates that the next bytes are a String, where the byte immediately following the $FF contains the length.

Propagated Label Cells

Propagated label cells are labels that place one particular ASCII character in each position of a window. Handy for visual effects like underlining.

+000	Flag Byte	Bit 7 is always zero. Bit 6 is meaningless. Bit 5 is always on. Bit 4 and bit 3 are protection, just like value cells. Bits 2, 1, and 0 are meaningless. Put a 1 here.
+001	Byte	This is the actual character that is to be put in each position in the cell.

Regular Label Cells

Regular label cells contain alphanumeric information, such as headings, names, and other descriptive information.

+000	Flag Byte	Bits 7, 6, and 5 are always zero. Bits 4 and 3 are same as value cells. Bits 2, 1, and 0 determine cell formatting: 01 Use spreadsheet standard formatting 02 Left justify 03 Right justify 04 Center
+001 to +nnn	Bytes	ASCII characters that actually display.

The actual length was defined earlier in the word that contained the actual number of bytes to read from disk.

File Tags

All AppleWorks files normally end with two bytes of $FF; tags are anything after that. Although File Tags were primarily designed by Beagle Bros, they can be used by any application that needs to create or modify an AppleWorks 3.0 file.

Because versions of AppleWorks before 3.0 stop at the double $FF, they simply ignore tags.

The File Tag structure is as follows:

+000	Byte	Tag ID. Should be $FF.
+001	Byte	2nd ID byte. These values will be defined and arbitrated by Beagle Bros Software. Beagle may be reached at: Beagle Bros Inc 6215 Ferris Square, #100 San Diego, CA 92121
+002 to +003	Word	Data length. If this is the last tag on the file, the low byte (+002) will be a count of the tags in this file, and the high byte (+003) will be $FF.
+004 to nnn	Bytes	Actual tag data, immediately followed by the next four-byte tag ID. These bytes do not exist for the last tag.

There is a maximum of 64 tags per file. Each tag may be no larger than 2K.

END OF FILE FTN.1B.xxxx

```
##########################################################################
### FILE: FTN.42.xxxx
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $42 (66)
Auxiliary Type: All

Full Name: File Type Descriptors
Short Name: File Type Names

Written by: Matt Deatherage **July 1989**

Files of this type contain File Type Descriptor tables.

Introduction

As applications continue to be assigned file type and auxiliary type combinations, the task of user file identification becomes more complex. If someone has the list in "About File Type Notes" memorized, simply displaying the file type and auxiliary type in hexadecimal is a suitable way of identification. However, few people have memorized this list. Programs such as the Finder have a need for this information in machine-readable form--not just a list of ASCII strings describing file types, but a way to take a given file's file type and auxiliary type and turn it into a string which describes the file. The Finder is not alone in this need, as parts of command shells, and sometimes entire programs, exist simply to identify files.

Developer Technical Support (DTS) has taken this opportunity to create a data structure that may be used by the Finder and any other application wishing to identify files. By using a separate file, the file identifiers can be updated between System Software releases, at the discretion of DTS, by releasing new descriptor files. Other applications may use the same file without having to reinvent the wheel. Furthermore, the multiple-file structure introduced and suggested in this Note allows developers to ship File Type Descriptor files with their software that allow the Finder and other applications to properly identify these files without a new release of the Finder (much as developers can supply their own Finder icons).

Note: Updated files, if and when released, will not result in changes being made to the System Software. The files as shipped with the System Software will remain unchanged until the next System Software release. Developer Technical Support reserves the right to release updated files for the convenience of those who wish to use them.

The File Format

The file's format is designed to give full and fast access to any file

descriptor string, while still remaining flexible enough to grow and be indicative of new features. Each file has three main parts: a header, an index, and the strings.

The Header

The file begins with a header which describes all the entries in the file:

+000	Version	Word	Version number. This is toolbox style, so revision x.y would be stored as $0xyy. x is the major revision number; when this value changes, it means that previously-written code will not be able to read this file. yy is the minor version number; when it changes, there are new fields but the old ones are in the same order. This Note describes version 1.0 of the File Type Descriptor format.
+002	Flags	Word	This word is all the flags words from all the records combined using a binary OR instruction. The flags word for each entry indicates the type of entry it contains (see the section "The Index Entries"). A particular bit in this word will be set if there exists a record in the file where the corresponding bit in the flags word is set. For example, bit 14 will be clear in this word if no index entry has bit 14 set.
+004	NumEntries	Word	The number of entries in this file.
+006	SpareWord	Word	Reserved for the application's use. The Finder calculates a value for each file and stores it in this field when the file is read into memory. This should be zero in the file on disk.
+008	IndexRecordSize	Word	The number of bytes in each index record.
+010	OffsetToIdx	Word	Offset, from the beginning of the file, to the first index entry. This field is provided so that other fields may be added to the header at a later date without breaking existing programs.

The Index Entries

The descriptions for each file type and auxiliary type assignment are pointed to by index entries for each string. If there is a string in the file that should be displayed for a particular assignment, there will be an index entry for it. If there is not an entry in any of the loaded files (see the section "Having More Than One File Type Descriptor File"), the string for file type $0000, auxiliary type $00000000 should be displayed.

The index contains one index entry for every file type and auxiliary type assignment or range (see below) in the descriptor file. All index entries in a given file are the same length (given in the header) so fast binary-searching algorithms may be performed. Their format is as follows for files with major version 1:

+000	Filetype	Word	The file type that should match for the string to which this index entry points.
+002	Auxtype	Long	The auxiliary type that should match for the string to which this index entry points.
+006	Flags	Flag Word	A word, defined bit-wise, indicating the type of match this entry contains. The following definitions apply if the bit in question is set: Bit 15: This record matches this file type and any auxiliary type. This bit would be set, for example, for a record for file type $FF (ProDOS 8 application). Bit 14: This record matches this auxiliary type and any file type. Bit 13: This record is the beginning of a range of file types and auxiliary types to match this string. Any file type and auxiliary type combination falling linearly between this record and the record with the same offset and bit 12 set should be given this string by default if no specific match is found. Bit 12: This record is the end of a range of file types and auxiliary types to match this string. Any file type and auxiliary type combination falling linearly between the record with the same offset and bit 13 set and this record should be given this string by default if no specific match is found. Bits 11-0: Reserved, must be set to zero when creating files.

Note: A range uses the file type and auxiliary type combined as a six-byte value, with the file type bytes as most significant. For example, file type $15, auxiliary type $4000 would fall in the range that starts with file type $13, auxiliary type $0800 and ends with file type $17, auxiliary type $2000

+008	Offset	Word	The offset, from the beginning of the file, to the Pascal string matching the description in this index entry. Note that more than one index entry can point to the same string.

The Strings

Since each index entry contains an offset to a string, it seems only logical that somewhere in the file is a string for each index entry. Apple recommends that the strings be placed in an array at the end of the index for most efficient use of space and ease in creating the file.

General Truths

So programs using File Type Descriptor files or resources don't have to construct all information about them each time they are opened, certain

characteristics will be true of all files. The following are characteristics which will always be true for files or resources with major revision number $01:

- The strings describing the files must each be no more than 30 characters long.
- The entries must always be sorted primarily by ascending file type and secondarily by ascending auxiliary type.
- Records that match file types or auxiliary types generically (for example, file type $FF and any auxiliary type) must contain zeroes for the wildcard field. A descriptor for ProDOS 8 application files should have file type $00FF, auxiliary type $00000000 and bit 15 set in the flags word. This record should be before any specific match for a file that has file type $FF and auxiliary type $0000, if such a record were to exist. Similarly, records which match a certain auxiliary type and any file type should appear before records which match file type $00 and that auxiliary type.
- The index entry marking the beginning of a range and the index entry marking the end of a range must not have any other index entries between them.
- Range index entries may not have bit 14 set.

Having More Than One File Type Descriptor File

More than one File Type Descriptor file may exist in a given directory. However, only one file may exist in a given directory with any auxiliary type from $00000000 to $000000FF. These files are provided by Apple Computer, Inc. and should not be altered by anything containing carbon atoms. Future implementations of System Software reserve the right to assume undocumented properties about File Type Descriptor files with auxiliary types smaller than $00000100. Editing of the strings in these files is not necessary, since other files may contain strings to override the ones in these files.

There is no such restriction on auxiliary types of $00000100 or greater.

To provide flexibility in changing file descriptions, applications should build in the capability to use as many File Type Descriptor files as are present. Files created by third-parties must follow the following two rules:

- The auxiliary type must not be lower than $00000100. Auxiliary types below $0100 are reserved for Apple.

- The File Type Descriptors must not contain a match for file type $00 and auxiliary type $0000. Such a descriptor contains the string to display for a file that does not match any other index entry. This entry must only be contained in the File Type Descriptor with auxiliary type zero.

A file with auxiliary type zero must exist. Others should be searched in order of descending auxiliary type, with $FFFFFFFF having highest priority. (This is why no file must contain a match for file type $00 and auxiliary type $0000 except the Apple-supplied one; otherwise, no searching would ever be done beyond the offending file.) In this way, strings in the Apple-supplied files may be superseded by other strings, without replacing or altering the Apple-supplied file (a feat that would be difficult anyway, due to the offset

nature of the file structure).

Program Use of More Than One File

Applications should search the directory for as many of the given files as can be found. If none is found with auxiliary type $0000, then the application should behave as if no files were found. When searching for a description, a separate search can be done on each file, stopping when a match is found. The search algorithm should return the "unknown" string when no specific match is found in the Apple-supplied file, so the search process will always return some string. An application should never run out of File Type Descriptors before finding a match.

Adding a File

Developers who wish to ship their own File Type Descriptor file with their product may contact Developer Technical Support for assistance in creating the file.

Memory Considerations

An application (especially a ProDOS 8 application) may not wish to spend valuable memory on files for file identification purposes, especially if directory listings are not an important part of the application. Since all offsets in the File Type Descriptor files are offsets from the beginning of the file, they may also be used with the ProDOS 8 or GS/OS SetMark call. Disk-based searches will obviously be much slower, but could be used for special instances (such as printing complete directories of disks as opposed to displaying them, or for specific functions that identify files).

About the Finder's Implementation

In Apple IIGS System Software 5.0, the Finder uses File Type Descriptor files. The Finder's implementation is somewhat limited, as this is a first pass at this new standard. The following implementation notes apply to Finder 1.3:

- The Finder looks for the File Type Descriptor files in the Icons directory of the boot disk (pathname *:Icons). It does not look in other directories or on other disks.
- Up to 30 File Type Descriptor files will be loaded.
- Two File Type Descriptor files are shipped with System Software 5.0. The first, FType.Main, contains file type descriptions for a small subset of file types, and no specific auxiliary types. This file will be loaded on machines with 512K or less of memory. The second file, FType.Aux, contains the rest of the descriptions shipped with System Software 5.0, as listed in "About Apple II File Type Notes" for July, 1989, and this file will be loaded in addition to the first on machines with more than 512K of memory. FType.Main has auxiliary type $0000; FType.Aux has auxiliary type $0001. The Finder does not depend on the names, but on the auxiliary types and file types.
- If the Finder cannot find any File Type Descriptor files in the *:Icons directory, it will terminate with fatal system error $4242. If it can not find a File Type Descriptor file with auxiliary type $0000, it will terminate with fatal system error $4243.

- The Finder will only use File Type Descriptor files with major version number $01. Also, the file will not be used if it any bits in the flags word of the header other than bit 15 are set, or if the spare word in the header is not zero, or if there are zero entries in the file. The Finder's search algorithm is fast, but currently does not handle special index entries other than for a given file type and any auxiliary type.

Further Reference

- About Apple II File Type Notes

END OF FILE FTN.42.xxxx

```
##########################################################################
### FILE: FTN.50.5445
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: **$50 (80)**
Auxiliary Type: **$5445**

Full Name: **Teach Document**
Short Name: **Teach document**

Written by: **Matt Deatherage** **March 1990**

Files of this type and auxiliary type contain TextEdit-based documents.

With the advent of TextEdit for the IIGS, many small text-editing applications have suddenly appeared. Apple may wish to release such a utility with the System Software in the future, and has defined this simple file format for the use of such a utility (referred to as "Teach" after the Macintosh program "TeachText" by Bryan Stearns) and the use of any other text-editing program.

The file format is eminently suitable for a Text Edit record, and applications should feel free to save any Text Edit record using this file format. The auxiliary type is the ASCII letters "TE".

The Data Fork

The data fork of a Teach document contains ASCII text. All 256 characters are allowed. The data fork should not contain anything that shouldn't be interpreted as ASCII text (for example, no data structures or pointers to lines, etc.).

The Resource Fork

The resource fork of a Teach document contains two resources. The first resource is of type rStyleBlock ($8012) and has ID $00000001. This resource contains all the style information for the text contained in the data fork. Each Teach document must contain this resource. Programs that use Teach documents may pass the resource ID $00000001 to TextEdit calls that require TEFormat data structures, which is the data structure contained in an rStyleBlock resource. If this resource does not exist, passing the ID to TextEdit could bring the system down.

The second resource is of type $7001 and has ID $00000001..This resource contains window position information. If the TextEdit record contains a ruler which is tied to the window position (so the text rewraps when the window is resized), when an application saves the record, it must include this resource. In the same respect, if this resource exists in a Teach file, an application must use this resource to set the window position and size.

Warning: If an application fails to follow either of these steps, TextEdit may hang.

The format of this second resource is as follows:

Height	(+000)	Word	The window height in pixels.
Width	(+002)	Word	The window width in pixels.
Top	(+004)	Word	The y coordinate of the window's top.
Left	(+006)	Word	The x coordinate of the window's left edge.
Version	(+008)	Long	Version number. Must be set to zero.

Other resources should not be placed in Teach documents. Any program saving a document could delete an older file with the same name instead of rewriting it, causing any other resources to be lost.

Further Reference

- Apple IIGS Toolbox Reference, Volume 3

END OF FILE FTN.50.5445

```
########################################################################
### FILE: FTN.50.8010
########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $50 (80)
Auxiliary Type: $8010

Full Name: AppleWorks GS Word Processor File
Short Name: AppleWorks GS Word Processor

Revised by: Matt Deatherage & Dave Lyons September 1990
Written by: Sydney R. Polk & Stephan Schwirzke, July 1990
Claris Corporation
Matt Deatherage, Apple Computer, Inc.

Files of this type and auxiliary type contain word processor documents from AppleWorks GS.
Changes since July 1990: Corrected the document format to include a count of the SaveArray entries, since such a count there is. Also added a description of how to count rulers.

AppleWorks GS is an integrated desktop productivity system for the Apple IIgs personal computer. AppleWorks GS includes a word processor, spreadsheet, data base, communications module and page layout module in one program. The word processor (whose file format is described herein) is paragraph-based and allows multiple fonts, styles, sizes, colors and rulers.

For more information on AppleWorks GS, contact:

Claris Corporation
5201 Patrick Henry Drive
P.O. Box 58168
Santa Clara, CA 95052-8168

Technical Support
Telephone: (408) 727-9054
AppleLink: Claris.Tech

Customer Relations
Telephone: (408) 727-8227
AppleLink: Claris.CR

Definitions

The following definitions are used in this document in addition to those defined for all Apple II file types:

Paragraph An AppleWorks GS paragraph consists of a paragraph header and a carriage return ($0D) with text in between. The paragraph header is defined later in this Note.

Font change A font change is signified by the one Byte token $01, followed by the Word new font family number.

Style change A style change is signified by the one Byte token $02 followed by the new Style Byte. The format of the Style Byte is included in this section.

Size change A size change is signified by the one Byte token $03, followed by the Byte new font size.

Color change A color change is signified by the one Byte token $04, followed by the Byte new color. The color is an offset (0-15) into QuickDraw II color table number zero.

Style Byte A style byte is a Byte of bit flags, defined as follows:

```
Bit 7:    Subscript
Bit 6:    Superscript
Bit 5:    Reserved for future use
Bit 4:    Shadow
Bit 3:    Outline
Bit 2:    Underline
Bit 1:    Italic
Bit 0:    Bold
```

When using a Style Byte with QuickDraw II, be sure to mask out bits 6 and 7 as QuickDraw II does not support these styles.

Text Block The Text Block is how AppleWorks GS stores text in memory. The format is as follows:

blockSize	(+000)	Word	The length of this Text Block, including the block size.
blockUsed	(+002)	Word	The number of bytes actually used by this Text Block. If this is less than blockSize, the remaining bytes should be ignored. This will not happen on disk.
theText	(+004)	Paragraphs	Paragraphs, as defined in this section.

Paragraphs are stored consecutively within Text Blocks, and a paragraph is not split over two or more Text Blocks. If there is more than one Text Block, consecutive Text Blocks contain consecutive sets of paragraphs.

Text Block Records

Text Block Records consist of a Long giving the size followed by a text block. A Text Block Record is redundant.

Reserved Characters

ASCII characters $01-$07 have special meaning in an AppleWorks GS WP file and are considered special characters.

Character $09 has is the Tab character and character $0D is the only paragraph ending character, the Return.

$01:	Font change	described in this section
$02:	Style change	described in this section
$03:	Size change	described in this section
$04:	Color change	described in this section
$05:	Page token	to be replaced with the page number of this page
$06:	Date token	to be replaced with the current ASCII date
$07:	Time token	to be replaced with the current ASCII time

Dates and times are in the "Thursday, July 5th, 1989 06:30 PM" format.

$09:	Tab	the tab character
$0D:	Return	ends a paragraph

Paragraph header

A paragraph header is seven bytes long:

firstFont	(+000)	Word	Font family number of the first character in the paragraph.
firstStyle	(+002)	Style Byte	The style of the first character in the paragraph.
firstSize	(+003)	Byte	The size (in points) of the first character in the paragraph.
firstColor	(+004)	Byte	The color of the first character in the paragraph, as an offset into QuickDraw II color table zero.
reserved	(+005)	Word	Reserved for future use.

Document header

A document header is found at the beginning of every AppleWorks GS word processing file. The header begins at offset zero and is 282 bytes long:

version	(+000)	Word	The version number of the file format. This is $1011 for AppleWorks GS version 1.0v2 and 1.1.
headerSize	(+002)	Word	Total size of the header in bytes. This is 282 ($11A) for version $0100.
refRecSize	(+004)	Word	Size of the reference record (fields rBits through rColor) in bytes. Always 48 ($30).
rBits	(+006)	22 Bytes	Each word in rBits is a bit flag representing the state of one of the AppleWorks GS menus when the file was saved. For example, if bit 0 of the fourth word is clear, then the first item in the fourth menu was disabled when the file was saved.

rUndo	(+028)	Long	Reserved; set to zero.
rState	(+032)	Long	Reserved; set to zero.
rNum	(+036)	Word	Reserved; set to zero.
rRefCon	(+038)	Long	Reserved; set to zero.
rChange	(+042)	Long	Reserved; set to zero.
rPrint	(+046)	Long	Reserved; set to zero.
rColor	(+050)	Long	Reserved; set to zero.
cTabSize	(+054)	Word	Size of the color table in bytes. This is always 64. This is twice as large as needed; the second 32 bytes of color table space are reserved for future expansion.
colorTable	(+056)	32 Bytes	The QuickDraw II color table for this document.
reserved	(+082)	32 Bytes	Reserved for future expansion. The size of this field is included in cTabSize.
pRecSize	(+120)	Word	Size of the print record in bytes. This is always 160.
printRecord	(+122)	160 Bytes	A Print Manager print record for this document.

Word Processor Global Variables

Some global variables for the document are calculated after the file is read; these are marked "reserved." The total size of the globals is 386 bytes. The document is actually stored as three documents--the text, the header and the footer, as is described in the "File Structure" section of this Note. The AppleWorks GS word processor swaps a section of data depending on whether the text, header or footer was showing when the file was saved. The first section reflects the state of the document at save time and is a duplicate of one of the other three sections, depending on the value of "stuff". The switched variables are defined in this section as "SwapVars."

intVersion	(+000)	Word	AWGS WP internal version; currently $0002.
view	(+002)	Word	The current view. Possible values are $0000 for the text, $0001 for the header and $FFFF (-1) for the footer.
stuff	(+004)	Word	Indicates which sections variables were swapped in when the file was saved. Possible values are $0000 for the text, $0001 for the header and $FFFF (-1) for the footer.
curDate	(+006)	String	The ASCII date when the file was saved. This field always takes 26

			bytes regardless of the length of the string.
curTime	(+032)	String	The ASCII time when the file was saved. This field always takes 10 bytes regardless of the length of the string.

Dates and times are in the "Thursday, July 5th, 1989 06:30 PM" format.

curPageNum	(+042)	String	The ASCII current page number (e.g., "15"). This field always takes 8 bytes regardless of the length of the string.

The next seven fields are used in headers and footers for time, date and page tokens.

docPages	(+050)	Word	Number of pages in current document.
startPage	(+052)	Word	Number with which to start pagination.
reserved	(+054)	Word	Reserved; set to zero when writing.
visRuler	(+056)	Word	Boolean; FALSE ($0000) if ruler is not showing, TRUE ($0001) if it is.
reserved	(+058)	Long	Reserved; set to zero when writing.
headerHeight	(+062)	Word	Height of header in pixels; maximum of 110.
footerHeight	(+064)	Word	Height of footer in pixels; maximum of 110.

The next 80 bytes are swapped out variables defined in this section:

currentVars	(+066)	SwapVars	80 bytes reflecting current variables when the document was saved.
docVars	(+146)	SwapVars	The document's variables.
headerVars	(+226)	SwapVars	The header's variables.
footerVars	(+306)	SwapVars	The footer's variables.

SwapVars

The SwapVars are variables that are different for the text, header and footer. The set of SwapVars in docVars is the variables at the time the file was saved. The remaining three sets of SwapVars apply to their sections of the file.

reserved	(+000)	Long	Reserved; set to zero when writing.
reserved	(+004)	Long	Reserved; set to zero when writing.
reserved	(+008)	Word	Reserved; set to zero when writing.

lastPrgph	(+010)	Word	The number of the last defined paragraph in the document. Paragraphs are numbered from one.
pageSize	(+012)	Word	Page size (vertically), in pixels.
topSpace	(+014)	Word	Top space above page, in pixels.
bottomSpace	(+016)	Word	Bottom space below page, in pixels.
paperSize	(+018)	Word	Paper size (vertically), in pixels.
horRulerRes	(+020)	Word	Horizontal resolution for ruler, in pixels
oPageRect	(+022)	Word	Offset from paper to page rect, horizontally, in pixels.
windPage	(+024)	Word	The page number that begins the current window.
lineOffset	(+026)	Word	How far down the top page the window starts, in pixels.
firstPrgph	(+028)	Word	Number of the paragraph (paragraphs are numbered from one) that has the first text on this page.
firstLine	(+030)	Word	Number of the first line in this paragraph in the window.
height	(+032)	Word	The height of the paragraph before the first line, in pixels in the window.
topSel	(+034)	Word	The paragraph number of the topmost portion of the selection, or zero for no selection.
topSelLine	(+036)	Word	The line number of the topmost portion of the selection.
selOffset	(+038)	Word	The offset into the paragraph in bytes of the first character of the selection.
reserved	(+040)	Long	Reserved; set to zero when writing.
insFlag	(+044)	Word	Zero for a single insertion point, one for a selected range.
caretEnd	(+046)	8 Bytes	End points of the caret line.
rangePar	(+054)	Word	The paragraph number of the end of the selection.
rangeLine	(+056)	Word	The line number of the end of the selection.
rangeOffset	(+058)	Word	The offset of the end of the selection
stylePending	(+060)	Boolean Word	

			TRUE if the current font has been changed but nothing has been typed.
fontID	(+062)	Long	The font ID of the current font.
color	(+066)	Word	The low byte is the current color byte (0-15); the high byte is zero.
topPrgphLine	(+068)	Word	The top paragraph on the screen.
topLine	(+070)	Word	The top line of the paragraph on the screen.
topPB	(+072)	Word	Top page boundary-- the page number of the top line in the window.
bottomPrgph	(+074)	Word	Paragraph number of the bottom paragraph on screen
bottomLine	(+076)	Word	Bottom line on screen
bottomPB	(+078)	Word	Bottom page boundary-- the page number of the bottom line in the window.

SaveArray entry

In the main document there will be one entry in a SaveArray for each paragraph in the document. Each entry is 12 bytes:

textBlock	(+000)	Word	Text Block number. Text Blocks are numbered from zero in the document; this entry shows in which text block this paragraph can be found.
offset	(+002)	Word	Adding this value to the offset of the text block gives the beginning of the paragraph.
attributes	(+004)	Word	$0000 = Normal text, $0001 = page break paragraph.
rulerNum	(+006)	Word	Number of the ruler associated with this paragraph. If this paragraph is a page break paragraph, ignore this field.
pixelHeight	(+008)	Word	Height of this paragraph in pixels.
numLines	(+010)	Word	Number of lines in this paragraph.

Ruler

Each paragraph has a ruler associated with it; the rulers are stored in the order in which they appear in the document and are numbered consecutively beginning with zero. Rulers are 52 bytes long and have the following structure:

numParagraphs	(+000)	Word	The number of paragraphs using this ruler.
statusBits	(+002)	Flag Word	Bits 15-8: Reserved for future use. Bit 7: Full justification

			Bit 6: Right justification Bit 5: Center justification Bit 4: Left justification Bit 3: Paragraph cannot break pages if this bit is set. Bit 2: Triple spaced (really double) Bit 1: Double spaced (really one and one half) Bit 0: Single spaced
leftMargin	(+004)	Word	Left margin in pixels from the left edge of the window.
indentMargin	(+006)	Word	Indent margin in pixels from the left edge of the window.
rightMargin	(+008)	Word	Right margin in pixels from the left edge of the window.
numTabs	(+010)	Word	This will be a number from one to ten; there is always at least one tab.
tabRecs	(+012)		10 Tab Records A tab record is defined in the following section.

Because rulers are defined consecutively from zero, you can use the SaveArray entries to find the total number of rulers. Look at the ruler number for each SaveArray entry; the highest-numbered ruler you find is an indication of the ruler count. For example, if the highest rulerNum in any SaveArray entry is $0003, there are four rulers in the document.

Tab Record A tab record identifies the type of tab in a ruler:

tabLocation	(+000)	Word	The location of the tab, in pixels, from the left edge of the screen.
tabType	(+002)	Word	The type of tab. $0000 is a left tab; $0001 is a right tab, and -1 ($FFFF) is a decimal tab, which centers around period characters.

File Format and Structure

The AppleWorks GS Word Processor file is composed of sections defined in "Definitions." The document structure is as follows:

docHeader	(+000)	Document Header	
globals	(+282)	WP Globals	
docSACount	(+668)	Word	Number of SaveArray entries to follow
docSaveArray	(+670)	SaveArray entries	One entry for each paragraph

docRulers	(+xxx)	Rulers	The rulers start here. "xxx" is at +670 + 12* the number of paragraphs.
docTextBlocks	(+yyy)	Text Block Records	The text block records start here. Note that when saved to disk, there is no extra space in a Text Block so the size is equal to the used field. "yyy" is at "xxx"+52*number of rulers.
headSaveArray		SaveArray entries	SaveArray entries for this document's header. The offset depends on the length of the document's text blocks.
headRulers		Rulers	The rulers for the header.
headTextBlocks		Text Block Records	The text for the header.
footSaveArray		SaveArray entries	The SaveArray entries for this document's footer.
footRulers		Rulers	The rulers for the footer.
footTextBlocks		Text Block Records	The text for the footer.

Please note that the number of paragraphs stored in the document is always one greater than the number displayed in the window. The last character is always a Return character. The number of carriage returns displayed is equal to the number of carriage returns stored minus one. This is so all of the paragraphs are guaranteed to end in a carriage return internally.

Blank document sections have a zero in the lastPrgph field of the SwapVars and have no save arrays, rulers, or text blocks.

The maximum number of paragraphs in a document is 64K-1 (65,535) and the maximum number of characters in a paragraph is 64K-13 (65,523), giving a maximum theoretical document size of a healthy 4,294,049,805 characters.

Further Reference

- Apple IIgs Toolbox Reference, Volumes 1 through 3

END OF FILE FTN.50.8010

```
##########################################################################
### FILE: FTN.53.8002
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $53 (83)
Auxiliary Type: $8002

Full Name: Graphic Disk Labeler Document
Short Name: Graphic Disk Labeler document

Written by: Matt Deatherage March 1990

Files of this type and auxiliary type contain label documents for Graphic Disk Labeler.

Graphic Disk Labeler is an Apple IIGS application which mixes text and graphics to create labels for 3.5" floppy disks. It imports most popular graphics formats and prints in color.

For more information on Graphic Disk Labeler (GDL), contact:

Triad Venture, Inc.
P.O. Box 12201
Hauppauge, New York 11788
Attention: GDL Technical Support
(516) 360-0797

File Structure

GDL documents contain the information for GDL to produce a label. A label is composed of three TextEdit records, a palette, and an optional picture. This information is in the data fork. The resource fork is reserved and should not be used.

The File Format

The data fork of GDL files contains the following data:

PicFlag	(+000)	Boolean Long	If this flag is TRUE, the next 5600 bytes contain a bit-mapped image of the graphic for this label. If this flag is FALSE, the next field is not present.
BitMap	(+004)	5600 Bytes	If PicFlag is TRUE, this is a bit-mapped image of this label's graphics. The rectangle is 100 pixels high by 104 pixels wide in 320 mode; this is also

		the size of the entire label. This field is not present if PicFlag is FALSE.

The remaining fields are present in every GDL document. They start at an offset referred to in this Note as "n". If there is no picture, "n" is 4; if there is a picture, "n" is 5604.

TERecGDL1	GDLText	GDL-style TextEdit record for the text on the back of the label.
TERecGDL2	GDLText	GDL-style TextEdit record for the text on the spine of the label.
TERecGDL3	GDLText	GDL-style TextEdit record for the text on the front of the label.
Palette	32 Bytes	Standard QuickDraw II Palette for this label.

The GDLText fields are defined as follows:

StyleLen	(+000)	Long	Length of TextEdit style information (TEStyle info) for this TextEdit record.
Style	(+004)	StyleLen Bytes	TextEdit style information. There are StyleLen bytes in this field.
TextLen	(+StyleLen+4)	Long	Length of the text in this TextEdit record.
Text	(+StyleLen+8)	TextLen Bytes	Text for this TextEdit record. There are TextLen bytes in this field.

Further Reference

- Apple IIGS Toolbox Reference Manual, Volumes 2 and 3

END OF FILE FTN.53.8002

```
##########################################################################
### FILE: FTN.54.DD3E
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: **$54 (84)**
Auxiliary Type: **$DD3E**

Full Name: **Medley Desktop Publishing Document**
Short Name: **Medley Document**

Written by: **Matt Deatherage & Eric Soldan** **May 1989**

Files of this type and auxiliary type contain documents for Medley(TM).

Medley is a WYSIWYG application that integrates word processing, paint, and page layout programs, with the addition of a spelling checker and thesaurus. The page layout function supports various shapes for art and text areas. Text automatically wraps around or within these areas, including irregularly shaped regions. The word processor is full-featured, as is the paint program. The dictionary has 80,000 words.

For more information on Medley, contact:

Milliken Publishing Company
1100 Research Blvd.
St. Louis, MO 63132
Attention: Medley Technical Support
(314) 991-4220

Definitions

The following definition is used in this document in addition to those defined for all Apple II file types:

C String **A series of ASCII bytes terminated with a byte of $00. There is no count byte at the beginning, as is the case for the String type (also referred to as a "Pascal string").**

File Structure

Medley files are basically standard, single-linked tree structures. There is a single object at the top of the tree, and other objects may branch off this parent object. Each child object is linked to the parent by a pointer to the child contained within the parent object. A non-standard thing about the Medley tree structure is that some objects may have regions or polygons associated with them. The handles to these objects are stored in the parent

object when in memory, but on disk these handles are quite meaningless. Because of this difference, the regions or polygons are simply appended to the parent object itself when written to disk. The size of the region or polygon is added to the size of the parent object, giving an aggregate size for the complex object on disk.

The file is written to disk in an order based on a simple tree-walking algorithm. This algorithm starts with the highest parent object and writes it to disk. The parent object is checked for child objects. If one exists, it is written to disk, and then it is checked for child objects. This tree-walking continues until an object runs out of children. When that occurs, Medley backs up one tree level, writes the next child object to disk, and scans it for children. This method continues until all objects are written to disk.

For example, if a parent object named A had two child objects named B and C, where B had children E and F, and C had children G and H, the objects would be written to disk in the following order: A, B, D, E, C, F, G. Figure 1 illustrates this structure.

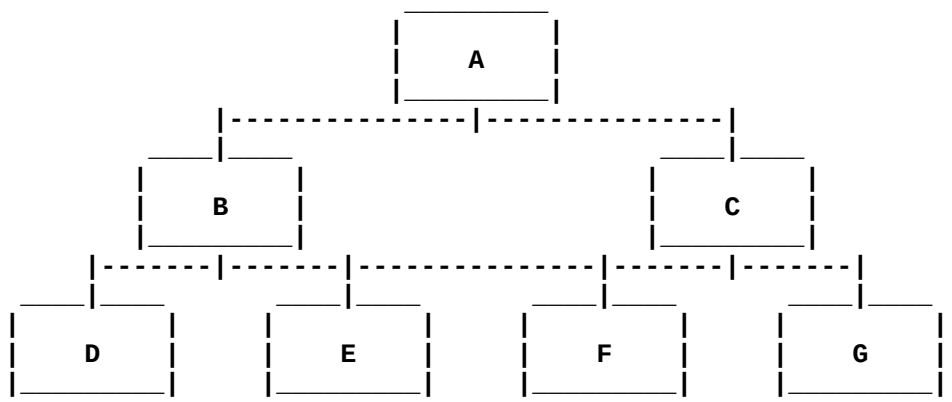


Figure 1-Example of Parent and Child Tree Structure

Some Medley objects, when in memory, have handles to other objects (such as regions or polygons) in them. Since handles are meaningless on disk, Medley stores these complex objects in an aggregate form by writing the contents of each associated handle to disk following the regular object.

The Objects and Their Formats

All objects have a common 13-byte header, which is as follows:

type	(+000)	Byte	The type of the object. Possible values are: 0 = Null Object (never saved to disk) 1 = Root Object (never saved to disk) 2 = File Object 3 = Page Object 4 = Paragraph Object 5 = Area Object 6 = Art Object 10 = Document Dictionary Object. Objects 7, 8 and 9 are for posting undo events. Since these are not saved to disk, they are irrelevant

			for the file format.
numChildren	(+001)	Word	The number of children for this object. Children of the children are not included.
endData	(+003)	Long	The size of this object on the disk, not counting associated handles (such as regions and polygons).
reserved	(+007)	Long	Reserved; set to zero when creating files.
objRefNum	(+011)	Word	The reference number of this object, generally zero.

Each object has the header listed above followed by object-specific data. For this reason, the description of each object will start with offset +013 (the byte following the header).

The File Object

rect	(+013)	4 Words	Standard QuickDraw II rectangle, giving the boundary rectangle for the entire file.
pathName	(+021)	129 Bytes	Class zero pathname for the file on disk (used by the save command).
saved	(+150)	Byte	This byte is 1 if no changes have been made to the file since the last save.
windowPtr	(+151)	Long	Pointer to the window for this file. When creating files, set to zero.
wndwNameIndx	(+155)	Byte	Index into table of window names. Set to zero when creating files
windowOrigin	(+156)	2 Words	QuickDraw II point representing the global origin of this file's window.
windowSize.h	(+160)	Word	Height of window in pixels. Add to top edge of window to get bottom edge of window.
windowSize.v	(+162)	Word	Height of window in pixels. Add to left of window to get right edge of window.
COrigin	(+164)	Long	QuickDraw II point representing the scroll bar origin of this file's window. When creating files, set this to whatever origin you wish Medley to display. Make sure that the coordinate is valid.
editHndl	(+168)	Long	Handle to the paragraph containing the cursor. This is converted to a child number on disk, since handles on disk are meaningless.
editOffset	(+172)	Word	Offset to the cursor within the paragraph pointed to by editHndl.
cursor	(+174)	4 Words	Standard QuickDraw II rectangle, giving the rectangle used for the insert cursor. This can be set to a

			null rectangle when creating files, and will be calculated when the file is loaded.
showAllBorders	(+182)	Byte	If this is set to one, then all area borders will display, regardless of each area's border display setting.
updateRect	(+183)	4 Words	Standard QuickDraw II rectangle used for posting specific updates for the interruptible word processor. When creating files, set this to a null rectangle.
topMrgn	(+191)	Fixed	The top margin in inches.
bottomMrgn	(+195)	Fixed	The bottom margin in inches.
leftMrgn	(+199)	Fixed	The left margin in inches.
rightMrgn	(+203)	Fixed	The right margin in inches.
gutterMrgn	(+207)	Fixed	The gutter margin in inches.
pageWidth	(+211)	Fixed	The width of the page in inches.
pageHeight	(+215)	Fixed	The height of the page in inches.
selectPage	(+219)	Word	The active page for area editing.
numSelected	(+221)	Word	The number of areas currently selected by the user.
sizingDot	(+223)	Word	The sizing dot number of an area that was last clicked. This sizing dot will be used as the current sizing dot when the arrows are used to size an area.
effectivePage	(+225)	Word	The page number of the page the user was effectively editing. This is different from selectPage when the user was editing global areas at save time, for global areas are treated as on page zero.
printRecord	(+227)	140 Bytes	A standard IIGS Print Manager print record; the one in use for this document at save time. This field can be undefined, if the printRecordDefined field is zero.
interruptMode	(+367)	Word	Currently undocumented. (Set to zero.)
editScroll	(+369)	Byte	Currently undocumented. (Set to zero.)
firstHndl	(+370)	Long	Handle where wrap-around regions should actually start; i.e., where an update is needed. When creating files, set this to zero.
firstMrn	(+374)	Word	MiniRect number (line number of paragraph) where wrap-around regions should actually start; i.e., where an update is needed. When creating files, set this to zero.
selectMode	(+376)	Word	Some text is selected if this is 1. Each paragraph will indicate the range of characters selected within that paragraph. This allows the screen update routine to quickly

			determine which characters in a paragraph should be drawn selected.
showPgphMarks	(+378)	Byte	Indicates whether paragraph marks are currently being shown.
showSpaces	(+379)	Byte	Indicates whether spaces are shown with marks and tabs are shown with arrows.
showMoveChangeInfo	(+380)	Byte	Indicates whether the Move/Change window is active.
moveChangeInfoRect	(+381)	4 Words	Standard QuickDraw II Rectangle giving the position of the Move/Change window.
addNewUndo	(+389)	Byte	Currently undocumented.
revNum	(+390)	Word	Revision number of the version of Medley that created this file. For files created following this standard, use $0100 (for Medley 2.0).
showRulers	(+392)	Byte	Indicates whether the rulers are showing.
windowType	(+393)	Word	The type of window this is. 0 = document, 1 = clipboard. When creating files, set this to zero.
auxDictPath	(+395)	129 Bytes	Class zero GS/OS pathname to the auxiliary dictionary file for this document. When creating files, set this to a null pathname (a length byte of zero).
grayScale	(+524)	Word	Whether or not grayScale mode is active. Zero for color, one for grayScale.
printRecordDefined	(+526)	Word	Non-zero if a print record is defined. This is used because the Printer and Port drivers must be loaded before calling PrDefault or PrValidate, and that means the boot disk must be on line. If Medley knows the print record is good, it proceeds without calling the Print Manager.
evenPageNumText	(+528)	48 Bytes	The text for even-numbered pages to be displayed by the page number.
oddPageNumText	(+576)	48 Bytes	The text for odd-numbered pages to be displayed by the page number.
pageNumInfo	(+624)	32 Words	A word for each of the pages possible in the file, through the absolute maximum of thirty-two.
affectPageRange	(+688)	2 Bytes	The range of pages affected by the Medley "Change Page Numbers" command. The first byte is the beginning page; the second byte is the ending page.
pageNumFont	(+690)	4 Bytes	A Font Manager FontID, identifying the font used for the

			page numbers. This is a two-byte font family number, followed by a one-byte font style and one-byte font size.
startPageNum	(+694)	Word	The page number of the first page. This is a zero-based counter; page one is represented as zero.
offsetFromEdge	(+696)	Word	The distance in points that the page numbering text appears from the edge of the paper.

Note: The following three fields are in Medley 2.0 files, but do not exist in Medley 1.0 files. If you are reading a 1.0 file, the value of revNum will be $0000. If reading a 1.0 File Object, resize it to 2.0 size (including the three fields below) and initialize their values to the values given below.

maxNumPages	(+698)	Word	The maximum number of pages in this document. When creating new files, initialize this to 32 ($20) unless condensed (below) is non-zero. If condensed is non-zero, you really have to hurt yourself to get this field right. Below is the algorithm Medley uses to calculate this field (in something close but not exactly related to pseudo-code). Please recall that all variables relating to the margins (taken from the file object) are Fixed.

```
workHeight := topMrgn + bottomMrgn
if [condensed is non-zero] then workHeight := workHeight * 2
workHeight := pageHeight - workHeight
workHeight := workHeight * [pixels per vertical inch]
workHeight := workHeight + $0000FFFF
[this counts a fractional point as a whole point]
i := HiWord(workHeight) [this gives the integer portion]
i := i + 3 [accounts for 3-pixel page breaks]
i := (16384 - 208) / i [gives number of pages in conceptual
drawing space.  Since Medley allows 48-point characters plus
leading, the tallest a text rectangle may be is 208 pixels.  Text
that does not fit in maxNumPages is kept around in a non-
displayable, non-editable, non-printable page.  Any shortening of
the document will cause some or all of the previously non-
displayed text to flow up into the document.]
i := min(32,i) [32 is the absolute maximum number of pages Medley
allows due to QuickDraw's conceptual drawing space limitations.]

maxNumPages := i
```

condensed	(+700)	Word	Indicates whether the document is designed to use condensed printing. If non-zero, the document is designed to use condensed printing. When creating files, it is easiest not to deal with condensed printing, so set this field to zero. However,

			if you wish to create a document that Medley may edit and print as condensed, you must correctly relate this field to the previous one by the algorithm given above.
reserved	(+702)	6 Bytes	These six bytes should be set to zero.

The Page Object

Pages are the first-level children of files. There is one page object for each page in a document (file object).

rect	(+013)	4 Words	Standard QuickDraw II rectangle giving the boundary rectangle for this page.
wrapDir	(+021)	Byte	The direction of word-wrapping. 1 = Down, 2 = across.
rgn	(+022)	Long	Handle to the region for this page in memory. The region is the page rectangle less any areas on that page. Global areas are not subtracted from this region. They are subtracted from the page rectangle for the global page (page zero). On disk, where the page region would be written, you can write a 10, followed by the rectangle for the page. This is a rectangular region. The aggregate size of the page object on disk must include these 10 bytes. This is assuming, of course, that there are no areas on that page to make the page region non-rectangular.
hideGlobalArt	(+026)	Byte	A non-zero value indicates that global art is not displayed on this page.
hideGlobalPageParts	(+027)	Byte	A non-zero value indicates that global page parts are not displayed on this page.

The Paragraph Object

Paragraphs are the children of the file object; they are not the children of page objects since a paragraph may be seen on more than one page or page part. Paragraph objects are, however, stored on disk immediately following page objects and their children. Paragraph objects are first-level objects also.

wrapHere	(+013)	Word	Insertion offset point in paragraph data where wrapping should continue. For wrapping from beginning of paragraph, set this field to zero.
fullWrap	(+015)	Word	Same as wrapHere, but indicates at what point miniRect construction or

			reconstruction must continue. Again, for full-wrapping of a paragraph, set to zero.
rulerOffset	(+017)	Word	Offset in bytes from beginning of paragraph object to indicate where the ruler starts. (The ruler is just before the character data, just after the miniRects.) If there is no ruler, then the default ruler is used by Medley. (If the dataOffset value is the same as the rulerOffset, then there is no ruler, and the default ruler will be used.) The default ruler has tabs at each 1/2 inch mark, no indent or paragraph indent, and the right margin is at maximum.
dataOffset	(+019)	Word	Offset in bytes from beginning of paragraph object to indicate where character data starts. If there are no miniRects built yet (probable if file is being created outside Medley) and there is no ruler, then this value will be a 32.
numRects	(+021)	Word	The number of discrete text rectangles in this paragraph. When creating a file, set fullWrap to zero, and numRects to zero, and place your character data starting at byte 32 of a paragraph object, and the rectangles will be built when the file is loaded by Medley as wrap occurs.
begInvOffset	(+023)	Word	Offset from the beginning of the character data where inverse text starts in this paragraph.
endInvOffset	(+025)	Word	Offset from the beginning of the character data where inverse text ends in this paragraph.
topLeading	(+027)	Byte	The number of pixels leading above each line in this paragraph.
botLeading	(+028)	Byte	The number of pixels leading below each line in this paragraphs.
begPgphGap	(+029)	Byte	The number of pixels extra leading above this paragraph.
endPgphGap	(+030)	Byte	The number of pixels extra leading below this paragraph.
flags	(+031)	Flag Byte	Bits 0 and 1 are used to indicate justify mode. 00 = left justify. 01 = right justify. 10 = center justify. 11 = full justify. Bit 7 indicates a page-break after this paragraph.
miniRects	(+032)	MiniRects	Any miniRects, if any, are contained here. The number of miniRects is given by numRects

above.

MiniRects have the following format:

miniRect.rect	(+000)	4 Words	Standard QuickDraw II rectangle that is calculated by the wordWrap routine to bound a line of text.
mr.begOffset	(+008)	Word	Offset from start of character data to the first character this miniRect bounds.
mr.endOffset	(+010)	Word	Offset from start of character data to just past the last character this miniRect bounds.

A Ruler in the document will be after the miniRects, if there are any. The offset to the Ruler is given by rulerOffset. Rulers are formatted as follows:

leftPgphMrgn	(+000)	Byte	The left margin for this paragraph, in sixteenths of an inch.
rightPgphMrgn	(+001)	Byte	The right margin for this paragraph, in sixteenths of an inch. This is an offset from the default right margin from Medley's "Set margins" command. For example, the value 16 represents a right margin one inch to the left of the default right margin.
pgphIndent	(+002)	Byte	The indentation for this paragraph, in sixteenths of an inch.
numTabs	(+003)	Byte	The number of tabs in this ruler.
tabs	(+004)	Tabs	There are numTabs of these.

Tabs are formatted as follows:

tab	Flag Word	Tabs consist of a high byte of flags and low byte of position. The bits are assigned as follows: Bits 15-12 = Reserved; set to zero. Bits 11-10 = Tab Leader style: 00 = No leader 01 = Leader of dots (.......) 10 = Leader of dashes (- -) 11 = Solid Leader (_____) Bits 9-8 = Tab Type: 00 = Left Tab 01 = Right Tab 10 = Center Tab 11 = Decimal Tab Bits 7-0 = Byte value; the position of this tab as an offset from the left margin in sixteenths of an inch. A value of sixteen indicates a tab one inch to the right of the left margin.

Following miniRects and rulers is the actual character data for this paragraph. This is all Bytes. However, a Byte value of $01 through $07 indicates the beginning of a Font Escape. Font Escapes indicate changes in style or size of the text, and are formatted as follows:

FontEscape	(+000)	Byte	An indication of the type of text the following fontID affects: 1 = Regular Text 2 = Superscript Text 3 = Subscript Text 4-7 = Reserved; do not use
fontID	(+001)	4 Bytes	A Font Manager FontID, identifying the font used for the page numbers. This is a two-byte font family number, followed by a one-byte font style and one-byte font size.

The text portion of a paragraph always begins with a Font Escape and ends with the end-of-paragraph character Byte $A6. This makes the minimum size of a paragraph (assuming no miniRects or rulers) thirty-eight bytes (32 bytes for the Paragraph Object, five bytes for the Font Escape and one byte for the $A6).

The Area Object

Area Objects are the children of pages or paragraphs.

type	(+013)	Byte	The type of area this area object describes. Possible values are: 0 = Null Area 1 = Group Area 2 = Rectangular Area 3 = Round Rectangular Area 4 = Oval Area 5 = Polygon Area
select	(+014)	Byte	This value is one if this area is selected.
showBorder	(+015)	Byte	This value is one if the border of this area is showing.
contentType	(+016)	Byte	0 = Art, 1 = Wrap Down, 2 = Wrap Across.
rgn	(+017)	Long	Handle to the region that describes the shape of this area. On disk, this region is at the end of this object (see the Reading The File section of this Note).
interiorRgn	(+021)	Long	Handle to the regions that describes the interior of this area. On disk, this region is at the end of this object (see the Reading The File section of this Note).
sizingRgn	(+025)	Long	Handle that contains all the sizing dots. It is too slow to draw them one at a time. Also, detecting

			that the user clicked in a sizing dot can be done quickly -- just not which one.
flags	(+029)	Word	Only bit zero of this word is significant; if set it indicates this area should be printed to LaserWriters in gray-scale. All other bits of this word should be zero.
reserved	(+031)	Word	Reserved for Milliken. Set to zero.

At this point is the description of the area itself. This description varies on the type field above:

For rectangles (type = 2):

rect	(+033)	4 Words	Standard QuickDraw II rectangle describing the rectangle for this area.

For round rectangle (type = 3):

rect	(+033)	4 Words	Standard QuickDraw II rectangle describing the boundary rectangle for the round rect.
height	(+041)	Word	The height of the oval portion of the rectangle.
width	(+043)	Word	The width of the oval portion of the rectangle.

For ovals (type = 4):

oval	(+033)	4 Words	A standard QuickDraw II rectangle describing the bounding rectangle for this oval. The oval drawn is the ellipse inscribed in this rectangle.

For Polygons (type = 5):

polygon	(+033)	Bytes	A handle to a QuickDraw II polygon. This handle may be passed to QD Polygon routines. On disk, this polygon is appended to the end of this object (see the Reading The File section on this Note).

These objects are the last items in the area object.

The Art Object

Art Objects are the children of pages, paragraphs or areas.

BBox	(+013)	4 Words	A standard QuickDraw II rectangle representing the bounding box of this art object.
offsetFromRgn	(+021)	2 Words	Normally zero. The area containing an art image can be grown

			and shrunk. The art within it is not clipped to the bounding rectangle of the area until the user deselects the area. (If it is saved to disk while selected, then it is saved unclipped). This allows the user to experiment with different shapes without clipping the drawing within. If the drawing is to the left of the left edge of the area, or is above the top edge, then this offset indicates by how much.
artImage	(+025)	Bytes	The actual bitmap of the art image.

The Document Dictionary Object

The Document Dictionary Object is the very last child of the file object, and contains all the words the spelling checker should ignore even though they are not in the main dictionary.

count	(+013)	Word	The number of word entries in this dictionary object.
wordList	(+015)	Word Entries	List of dictionary word entries.

The format of word entries is as follows:

recordLength	(+000)	Byte	The length of this record.
replaceFlag	(+001)	Byte	Reserved, set to zero.
newWord	(+002)	C String	The word in question. This word should be counted as spelled correctly, and is not in the Main or Auxiliary Dictionary.

The length of a record is the length of string plus three bytes (one for recordLength, one for replaceFlag, and a zero termination byte).

Reading the File

When reading a Medley file, objects with regions or polygons will have to be treated specially, since the handles in the objects are invalid and the regions or polygons actually follow the object in the disk file.

A sequence for reconstructing Medley files in memory is as follows:

1. Open the file, or set the mark to zero on an open file.
2. Start with a handle that is 13 ($0D) bytes long. Pass this handle to the routine starting in step three.
3. Save the handle passed to this routine, and read four bytes from disk. This Long is the total size of an object, including any regions or polygons appended.
4. Read the 13-byte object header into the handle passed to this routine. The endData field of the header gives the size of the object, minus any associated regions or polygons. Resize the object's handle (the handle passed to you) to this size.

5. Read the rest of the object (endData - 13 bytes) into the object's handle.
6. Save the value of numChildren in a local variable and set the numChildren field in the object header to zero. The field in the header represents the number of children read from disk; setting this to zero properly indicates that you haven't read any of the children yet.
7. Look at the object type field. If the object is a file, area, or page object, it may have a region associated with it. If the object is an area object, it may also have a polygon associated with it (if the area type field indicates this is so). You can tell if the object has any appended structures by comparing the total object size (read in step three) with the endData field (read in step four); if an object has no appended structures, the two values will be the same.

   If there are structures appended to the object, first zero all the handles to the regions inside the object. This allows elegant error recovery if an error occurs while reading the region or polygon. When the handles are zeroed, read the next two bytes from the disk. This Word is the size of the region or polygon in bytes. Create a handle of that size, place it in the object's field for this handle, and place the size Word in the first two bytes of the new handle. Now read the object from disk into the new handle starting at the beginning +002 (past the size Word).

   Continue in this fashion until all appended regions or polygons have been read from disk. Any appended structures will be stored in the same order as their handles occur in the object.

   Note: By zeroing the handles before reading the objects, you can return from this function with an error, and the calling routine will be able to dispose of all handles that were actually created. The calling routine will know if a handle was actually created or not by examining the handle field in the object; NIL handles were not created.

8. Execute a loop for the old number of children (0 to oldNumChildren-1):
9. Create an object that is 13 ($0D) bytes long. Add this handle to the end of the parent object that was last read. Increment the number of children. You have just added a child into the child table for an object.
10. Call the recursive subroutine beginning in step three, passing it the handle you just created. If it returns an error, return the error. This gets you out of the recursion with the correct error, no matter how many levels deep you are.
11. Keep looping until out of children to read. The EOF condition does not have to be checked, since you will run out of children when you reach the end of the file. If an EOF is reached before you read all the children, you did something wrong.
12. Return no error--the file was successfully read.
13. When done with all this, you will return to the code just beyond step two, where you first called the recursive subroutine at step three. If an error is returned, dispose of all the handles created by the recursive function. Even if the file read is aborted, the tree is complete for as much as was read.

(This is why the numChildren field is incremented as you read the file.) An alternate way to handle this is to use a different userID for the handles created when reading the file; this allows you to dispose of all of them with one DisposeAll call.

14. Close the file if you opened it, or reset the mark to its previous position if it was already open.

The entire file does not have to be read from disk. By using the size field, you can skip to the next object in the file. Using this technique, you can scan the file for whatever it is that interests you.

Note: You may have noticed that objects successfully created in memory will have a table of handles to children at the end. Objects on disk will not have these handles, since the handles on disk are meaningless. The child handle table is reconstructed as the file is loaded into memory.

Object Ordering

The file object is the first you will encounter in a Medley file. Its children are ordered as follows:

Page Object--Page #0. This is the global page object, containing all global areas.
Page Object--Page #1. This is the page object for page #1; it must exist.

Other objects are optional, but will appear in the following order:

Page #2 through Page #n
Paragraph #1 through Paragraph #n
Dictionary Object

Some Example Structures

Medley was written mostly in C. Below are some structures relevant to C programs reading Medley files. Descriptions of the fields may be found earlier in this Note.

```
#define NULLOBJ       0    /* Object type assignments. */
#define ROOTOBJ       1    /* These are used in the deskObj 'types' field. */
#define FILEOBJ       2
#define PAGEOBJ       3
#define PGPHOBJ       4
#define AREAOBJ       5
#define ARTOBJ        6
#define DOCDICTOBJ    10

#define AREANULL      0    /* Area object sub-type assignments. */
#define AREAGROUP     1    /* These are used in the areaObj 'types' field. */
#define AREARECT      2
#define AREARRECT     3
#define AREAOVAL      4
#define AREAPOLY      5
```

```
#define medleyMainType    0x54
#define medleyAuxType     0xDD3E
#define medleyInfo        1
#define auxDictType       2


#define ARTCONTENT    0    /* These are used in the 'contentType' field of
                              area objects.*/
#define WWDOWN        1
#define WWACROSS      2
#define LWGRAYSCALE   0x0001


#define SAMEESC       0    /* These are used in paragraph objects. */
#define FONTESC       1
#define SUPERESC      2
#define SUBESC        3
#define ESCAPES       7


#define SIZEFONTESC   5    /* More paragraph equates. */
#define ENDPGPHCHR    0xA6
#define TABCHR        9
#define SOFTHYPHEN    30
#define STICKYSPACE   31


#define PAGEBREAK     0x80  /* These are used in the pgphObj 'flags' field. */
#define LEFTJUST      0x00
#define RIGHTJUST     0x01
#define CENTERJUST    0x02
#define FULLJUST      0x03
#define JUSTTYPES     0x03

#define LEFTTAB       0x00  /* These are used in the ruler field of paragraph
                              objects.  */
#define RIGHTTAB      0x01
#define CENTERTAB     0x02
#define DECIMALTAB    0x03
#define TABTYPES      0x03
#define NOLEADER      0x00
#define DOTSLEADER    0x01
#define DASHESLEADER  0x02
#define SOLIDLEADER   0x03

typedef struct Ruler {
       unsigned char    leftPgphMrgn;
       unsigned char    rightPgphMrgn;
       unsigned char    pgphIndent;
       unsigned char    numTabs;
       unsigned int     tab[];
} Ruler;

#define NEWREVNUM 0x0100


typedef union URect {
       Rect     rect;
```

```
        struct {
                long    p1;
                long    p2;
        } point;
        struct {
                Point   p1;
                Point   p2;
        } ele;
} URect;

typedef union UPoint {
        Point   ele;
        long    point;
} UPoint;

typedef struct region {
        unsigned int    size;
        union URect     BBox;
        int             data[];
} region;

typedef struct polygon {
        int             size;
        union URect     BBox;
        union UPoint    point[];
} polygon;

typedef union ourFontID {
        unsigned long    fid;
        struct {
                unsigned int    famNum;
                char            fontStyle;
                char            fontSize;
        } f;
} ourFontID;

struct deskObj {
      char              type;           $00
      unsigned int      numChildren;    $01
      unsigned long     endData;        $03
      unsigned long     reserved;       $07
      unsigned int      objRefNum;      $0B
                                        $0D

    union d {
        data[];           /*  Plain label object access field.  */

        struct file {           /*  Level 1 objects are files.  */
              union URect       rect;                   /* $0D */
              char              pathName[129];          /* $15 */
              char              saved;                  /* $96 */
              GrafPortPtr       windowPtr;              /* $97 */
              char              windowNameIndx;         /* $9B */
              long              windowOrigin;           /* $9C */
              long              windowSize;             /* $A0 */
              long              COrigin;                /* $A4 */
              struct deskObj    **editHndl;             /* $A8 */
              unsigned int      editOffset;             /* $AC */
```

```
      union URect       cursor;                 /* $AE */
      char              showAllBorders;         /* $B6 */
      union URect       updateRect;             /* $B7 */
      unsigned long     topMrgn;                /* $BF */
      unsigned long     bottomMrgn;             /* $C3 */
      unsigned long     leftMrgn;               /* $C7 */
      unsigned long     rightMrgn;              /* $CB */
      unsigned long     gutterMrgn;             /* $CF */
      unsigned long     pageWidth;              /* $D3 */
      unsigned long     pageHeight;             /* $D7 */
      int               selectPage;             /* $DB */
      int               numSelected;            /* $DD */
      int               sizingDot;              /* $DF */
      int               effectivePage;          /* $E1 */
      PrRec             printRecord;            /* $E3 */
      int               interruptMode;          /* $16F */
      char              editScroll;             /* $171 */
      struct deskObj    **firstHndl;            /* $172 */
      int               firstMrn;               /* $176 */
      unsigned int      selectMode;             /* $178 */
      char              showPgphMarks;          /* $17A */
      char              showSpaces;             /* $17B */
      char              showMoveChangeInfo;     /* $17C */
      union URect       moveChangeInfoRect;     /* $17D */
      char              addNewUndo;             /* $185 */
      unsigned int      revNum;                 /* $186 */
      char              showRulers;             /* $188 */
      unsigned int      windowType;             /* $189 */
      char              auxDictPathname[129];   /* $18B */
      unsigned int      grayScale;              /* $20C */
      unsigned int      printRecordDefined;     /* $20E */
      char              evenPageNumText[48];    /* $210 */
      char              oddPageNumText[48];     /* $240 */
      unsigned int      pageNumInfo[MAXNUMPAGES];/* $270 */
      unsigned int      affectPageRange;        /* $2B0 */
      ourFontID         pageNumFont;            /* $2B2 */
      unsigned int      startPageNum;           /* $2B6 */
      unsigned int      offsetFromEdge;         /* $2B8 */
      unsigned int      maxNumPages;            /* $2BA */
      unsigned int      condensed;              /* $2BC */
      char              reserved[6];            /* $2BE */
} file;                                         /* $2C4 */

struct page {
      union URect       rect;                   /* $0D */
      char              wrapDir;                /* $15 */
      region            **rgn;                  /* $16 */
      char              hideGlobalArt;          /* $1A */
      char              hideGlobalPageParts;    /* $1B */
} page;                                         /* $1C */

struct pgph {          /*  Must be level 2 or greater.  */
      unsigned int      wrapHere;               /* $0D */
      unsigned int      fullWrap;               /* $0F */
      unsigned int      rulerOffset;            /* $11 */
      unsigned int      dataOffset;             /* $13 */
      unsigned int      numRects;               /* $15 */
      unsigned int      begInvOffset;           /* $17 */
```

```
            unsigned int      endInvOffset;           /* $19 */
            char              topLeading;             /* $1B */
            char              botLeading;             /* $1C */
            char              begPgphGap;             /* $1D */
            char              endPgphGap;             /* $1E */
            char              flags;                  /* $1F */
          struct miniRect {                           /* $20 */
                union URect     rect;
                unsigned int    begOffset;
                unsigned int    endOffset;
          } miniRect[];
/*    Ruler goes here if there is a custom ruler for this paragraph.*/
/*    Text starts after the ruler.  Text always
      starts with a fontEsc.  A fontEsc is 5 bytes,
      a typeByte followed by the fontID.  Text
      always ends with end-of-pgph chr. */
        } pgph;                                       /* $20 */

        struct area {
            char              type;                   /* $0D */
            char              select;                 /* $0E */
            char              showBorder;             /* $0F */
            char              contentType;            /* $10 */
            region            **rgn;                  /* $11 */
            region            **interiorRgn;          /* $15 */
            region            **sizingRgn;            /* $19 */
            unsigned int      flags;                  /* $1D */
            unsigned int      reserved;               /* $1F */
          union obj {
              union  URect    rect;                   /* $21 */
              struct rrect {
                     union URect     rect;            /* $21 */
                     int             height;          /* $29 */
                     int             width;           /* $2B */
              } rrect;
              union  URect    oval;                   /* $21 */
              polygon         **poly;                 /* $21 */
          } obj;
        } area;                                       /* $2D */

        struct art {
            union URect     BBox;                     /* $0D */
            union UPoint    offsetFromRgn;            /* $15 */
            char            artImage[];               /* $19 */
        } art;

        struct docDict {
            unsigned int    count;                    /* $0D */
            char            wordList[]                /* $0F */
        } docDict;                                    /* $0F */

    } d;
};

### END OF FILE FTN.54.DD3E
```

```
##########################################################################
### FILE: FTN.5A.0000
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $5A (90)
Auxiliary Type: $0000
Pathname: *:System:Sounds:Sound.Settings

Written by: Dave Lyons May 1992

The file with this type, auxiliary type, and pathname records the user's mappings from SysBeep2 codes to sound names.

The File Format

The file lists a number of Word/String pairs, just like an rTaggedStrings resource:

```
count        (+000) Word     Number of word/string pairs.
firstWord    (+002) Word     Word value of first pair.
firstString  (+004) String   Pascal string of first pair.
secondWord   (+xxx) Word     Word value of second pair.
secondString (+yyy) String   Pascal string of second pair.
...          ...    ...      ...
```

The word in each pair is either a SysBeep2 code ($0000..$3FFF) or $FFFE (for System Beep). The string in each pair is the resource name of an rSoundSample resource present in any file in the *:System:Sounds folder.

There are two special strings: "*" means Standard Beep, and "0" (zero) means Silence.

The order of the pairs is not important, but no two pairs can have the same word value.

Further Reference

- System 6 Documentation

END OF FILE FTN.5A.0000

```
##########################################################################
### FILE: FTN.5A.0002
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: **$5A (90)**
Auxiliary Type: **$0002**

Full Name: **Battery RAM saved configuration**
Short Name: **Battery RAM configuration**

Written by: **Matt Deatherage** **May 1990**

Files of this type and auxiliary type contain images of Apple IIgs Battery RAM.

Since the IIgs battery-powered RAM contains important system parameters and control information, many utility programs would like to save an image of all these parameters for later restoration. This ability can be handy when changing batteries or in a classroom situation where students may change text colors and preferences to suit their individual tastes--without regard for the teacher who has to restore the machine to normal operation for the next class.

Apple has defined the following simple file format for use in such instances.

The File Format

The file format contains a safety catch to prevent parameters defined for new machines from interfering with older machines.

ROMVersion	**(+000)**	**Word**	**The version number of the Apple IIgs ROM for the machine on which this file was created. The file should only be restored on machines of the same ROM version. For example, if this value is $0003, the image of Battery RAM should not be written to a ROM 01 machine. Instead, warn the user of the incompatibility.**
BRAM	**(+002)**	**256 Bytes**	**256 bytes of battery RAM image. This is obtained from the ReadBRam tool call and should be restored (where appropriate) with the WriteBRam tool call.**

Further Reference

- **Apple IIgs Firmware Reference**
- **Apple IIgs Toolbox Reference, Volume 1**

END OF FILE FTN.5A.0002

```
##########################################################################
### FILE: FTN.5A.802F
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $5A (90)
Auxiliary Type: $802F

Full Name: Cool Cursor document
Short Name: Cool Cursor document

Written by: Josef W. Wankerl & Matt Deatherage May 1992

Files of this type and auxiliary type contain cursor resources for Cool Cursor.

Cool Cursor is a commercial Control Panel available from GS+ Magazine. It replaces the normal watch cursor with an animated cursor defined by the rCursor resources in a Cool Cursor document.

For more information on Cool Cursor or GS+ Magazine, contact:

GS+ Magazine
P.O. Box 15366
Chattanooga, TN 37415-0366
Attention: Cool Cursor Technical Support
(615) 843-3988

America Online: GSPlusDiz
Delphi: GSPlusDiz
GEnie: JWANKERL
Internet: jwankerl@pro-gonzo.cts.com

FILE FORMAT

A Cool Cursor document is an extended file with an empty data fork. The resource fork must contain an rPString ($8006) resource with an ID of $00000001, which is the name of the cursor. The cursor name is displayed in a list and must uniquely identify the cursor (i.e. there cannot be two cursors named "Beachball.") To make sure your cursor's name will not conflict with existing cursor names you may contact GS+ Magazine.

There are two lists of rCursor ($8027) resources--one list for 640 mode and one list for 320 mode. Both lists are optional, and if absent the cursor will default to the standard QuickDraw Auxiliary WaitCursor. The 640 mode rCursor list is a contiguous set of resources that starts at resource ID $00001000 and can increase to $00001FFF. The 320 mode rCursor list is a contiguous set of resources that starts at resource ID $00002000 and can increase to $00002FFF.

For example, a Cool Cursor document called "Beachball" that contained a two-frame animation for both 640 and 320 modes would have resources as shown

in Table 1:

Type name	Resource ID	Resource Contents
rPString	$00000001	Beachball
rCursor	$00001000	the first 640 mode cursor frame
rCursor	$00001001	the second 640 mode cursor frame
rCursor	$00002000	the first 320 mode cursor frame
rCursor	$00002001	the second 320 mode cursor frame

Table 1--Cool Cursor Example Resource List

Further Reference

- Apple IIgs Technical Note #76, Miscellaneous Resource Formats

END OF FILE FTN.5A.802F

```
##########################################################################
### FILE: FTN.5A.8031
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $5A (90)
Auxiliary Type: $8031

Full Name: Replicator preferences
Short Name: Replicator preferences

Written by: Josef W. Wankerl & Matt Deatherage May 1992

Files of this type and auxiliary type contain preferences for the disk duplicating application Replicator.

Replicator is a commerical, desktop-based disk duplicating application available from GS+ Magazine.

For more information on Replicator or GS+ Magazine, contact:

GS+ Magazine
P.O. Box 15366
Chattanooga, TN 37415-0366
Attention: Replicator Technical Support
(615) 843-3988

America Online: GSPlusDiz
Delphi: GSPlusDiz
GEnie: JWANKERL
Internet: jwankerl@pro-gonzo.cts.com

FILE FORMAT

A Replicator preferences file is an extended file with an empty data fork. Through version 1.1, Replicator looks for a file named "ReplicatorPrefs" in the "@" directory and gets preferences from that file if it exists. The resource fork should contain eight resources of type $0001, with resource IDs $00000001 through $00000008.

Resource IDs $00000001 through $00000005 are WORD values which correspond to checkboxes in Replicator's preferences dialog. If the word is zero, the checkbox is unchecked, while non-zero values indicate the checkbox is checked.

Resource IDs $00000006 through $00000008 are WORD values which correspond to numeric values in Line Edit controls in Replicator's preferences dialog. The exact correspondence for these eight values is shown in Table 1.

ID	Preference

$00000001	Duplicate on exact size devices only
$00000002	Format media only when necessary
$00000003	Prompt on formatted target disks
$00000004	Blank screen on formats and writes
$00000005	Clear errors after disk inserts
$00000006	Verify disk blocks every nth disk
$00000007	Validate disk files every nth disk
$00000008	Compare disk images every nth disk

Table 1--ID to Preference Mapping

As an example, a Replicator preferences file that has "Duplicate on exact size devices only" turned off, "Format media only when necessary" turned on, "Prompt on formatted target disks" turned on, "Blank screen on formats and writes" turned off, "Clear errors after disk inserts" turned on, "Verify disk blocks every 10 disks," "Validate disk files every zero disks" and "Compare disk images every 1 disks" would contain the resources listed in Table 2.

Resource Type	Resource ID	Contents
$0001	$00000001	$0000
$0001	$00000002	$0001
$0001	$00000003	$0001
$0001	$00000004	$0000
$0001	$00000005	$0001
$0001	$00000006	$000A
$0001	$00000007	$0000
$0001	$00000008	$0001

Table 2--Replicator Example Resource List

END OF FILE FTN.5A.8031

```
##########################################################################
### FILE: FTN.5A.XXXX
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: **$5A (90)**

Full Name: **Application Configuration file**
Short Name: **Configuration file**

Written by: **Matt Deatherage** **May 1992**

Files of this type and auxiliary type contain configuration or preferences.

Files of type $5A contain configuration information or preferences for Apple II software. The original name for this file type is "Configuration," but it's friendlier to call such application-specific settings "preferences." It's easier for most people to understand--most people don't "configure" things often, but everyone has their own preferences.

Any program can have preferences, although it's most common for applications. However, inits, DAs and other system components can have preferences as well--see the DTS Sample Code "IR 2.0.1" for an example of an init with a preferences file.

Apple strongly recommends requesting a preferences auxiliary type if you create preferences files, so that other programs may identify your files (even if they don't operate on them). However, if your files are named something very self-explanatory (such as the Sound Control Panel's "Sound.Settings" file), you may use auxiliary type $0000 provided you always identify your files by file name.

Preferences files belong in the same directory as the program that owns them unless the program is on a network file server, in which case the preferences should go in the user folder. If you don't use the user folder, each user has to have write permission to the application folder (not always practical), and even so all users would have to use the same preferences. For GS/OS applications, proper preference management is easy--using the "@" prefix puts your files exactly where they should go. It's a little trickier for non-applications, but the AppleShare-specific call GetUserPath makes it not too difficult. The previously-mentioned Sample Code (IR 2.0) shows how an init written in 65816 assembly language can place preference files properly.

Even if you do this, for registration and future identification purposes, we recommend you get an auxiliary type assignment. If you intend to identify your files by name and give them a self-explanatory name, you may request with your assignment that no custom File Type Descriptor string be included with the system software, which will save disk space and memory.

Note that using auxiliary type $0000 when identifying preferences by name is an exception to the normal rule--in other file types, you may NOT use

auxiliary type $0000.

END OF FILE FTN.5A.XXXX

```
##########################################################################
### FILE: FTN.5E.0001
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $5E (94)
Auxiliary Type: $0001

Full Name: Resource file
Short Name: Resource file

Written by: Matt Deatherage May 1990

Files of this type and auxiliary type contain resources of various types.

Files of type $5E and auxiliary type $0001 contain resources in an Apple IIgs Resource Manager compatible resource fork. The presence or absence of any particular type or ID of resource cannot be assumed--the file simply contains zero or more resources. The data fork is reserved and should not be used.

These files are typically used by resource editors, movers and other resource utilities. Newly-created files should be of type $5E and auxiliary type $0001, unless a more specific assignment is necessary for a particular application. Developer Technical Support assigns auxiliary types in those instances as necessary.

Further Reference

- Apple IIgs Toolbox Reference, Volume 3

END OF FILE FTN.5E.0001

```
##########################################################################
### FILE: FTN.A0.0000
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: **$A0 (160)**
Auxiliary Type: **$0000**

Full Name: **WordPerfect Word Processing File**
Short Name: **WordPerfect Document**

Written by: **Matt Deatherage** **January 1989**

Files of this type and auxiliary type contain WordPerfect(TM) documents.

WordPerfect(TM) for the Apple IIe and IIc provide more than 110 word processing features. In addition to a 50,000-word spelling checker and a thesaurus, the program offers footnotes, headers and footers, mail-merge, macros, search and replace, on-line help, file management, and dozens of formatting features.

WordPerfect for the Apple IIGS includes all these features and more. The program has a mouse interface with pull-down menus and allows for the editing multiple documents. Users can edit an unlimited number of documents at the same time and can take advantage of any memory available.

For more information on WordPerfect, contact:

WordPerfect, Inc.
1555 North Technology Way
Orem, Utah 84057
Attention: Apple II WordPerfect Technical Support
Telephone: (801) 225-5000

File Structure

WordPerfect files are ASCII files. The text is neither encrypted nor compacted. There is no end-of-file character, and padding past the end of a document with garbage may cause WordPerfect to crash.

Formatting or function codes are embedded as they occur in the text. No code is used for beginning of file or end of file. When creating WordPerfect files from other programs, keep in mind the following:

- **The initial WordPerfect margin settings are usually 10 and 74, and it is best to keep the line length to 65 or less, unless you change the margin settings.**
- **It is best not to pad with spaces.**

Function Codes

All function codes are listed in hexadecimal. All except the first five have the high bit set. Function codes above $C0 are discussed under "Multi-Byte Functions."

Code	Command
$09	Tab
$0A	Hard New Line
$0B	Soft New Page
$0C	Hard New Page
$0D	Soft New Line
$80	No operation
$81	Turn right justification on
$82	Turn right justification off
$83	End of centered text
$84	End of aligned or flushed right text
$85	(Used in other WordPerfect Corporation Products)
$86	Center page from top to bottom
$87	(Used in other WordPerfect Corporation Products)
$88	(Used in other WordPerfect Corporation Products)
$89	Tab after the right margin
$8A	Widow/orphan on
$8B	Widow/orphan off
$8C	Hard end of line and soft end of page
$8D	Footnote number (appears only inside of footnotes)
$8E	Reserved
$8F	Reserved
$90	(Used in other WordPerfect Corporation Products)
$91	(Used in other WordPerfect Corporation Products)
$92	(Used in other WordPerfect Corporation Products)
$93	(Used in other WordPerfect Corporation Products)
$94	Underline on
$95	Underline off
$96	Reverse video on
$97	Reverse video off
$98	(Used in other WordPerfect Corporation Products)
$99	Overstrike
$9A	Cancel hyphenation of following word
$9B	(Used in other WordPerfect Corporation Products)
$9C	Bold off
$9D	Bold on
$9E	Hyphenation off
$9F	Hyphenation on
$A0	Hard space
$A1	(Used in other WordPerfect Corporation Products)
$A2	(Used in other WordPerfect Corporation Products)
$A3	(Used in other WordPerfect Corporation Products)
$A4	(Used in other WordPerfect Corporation Products)
$A5	(Used in other WordPerfect Corporation Products)
$A6	(Used in other WordPerfect Corporation Products)
$A7	(Used in other WordPerfect Corporation Products)
$A8	(Used in other WordPerfect Corporation Products)
$A9	Hard hyphen in line
$AA	Hard hyphen at end of line

$AB	Hard hyphen at end of page
$AC	Soft hyphen
$AD	Soft hyphen at end of line
$AE	Soft hyphen at end of page
$AF	(Used in other WordPerfect Corporation Products)
$B0	(Used in other WordPerfect Corporation Products)
$B1	(Used in other WordPerfect Corporation Products)
$B2	(Used in other WordPerfect Corporation Products)
$B3	(Used in other WordPerfect Corporation Products)
$B4	(Used in other WordPerfect Corporation Products)
$B5	(Used in other WordPerfect Corporation Products)
$B6	(Used in other WordPerfect Corporation Products)
$B7	(Used in other WordPerfect Corporation Products)
$B8	(Used in other WordPerfect Corporation Products)
$B9	(Used in other WordPerfect Corporation Products)
$BA	(Used in other WordPerfect Corporation Products)
$BB	(Used in other WordPerfect Corporation Products)
$BC	Superscript
$BD	Subscript
$BE	Advance 1/2 line up
$BF	Advance 1/2 line down

Multi-Byte Functions

Multi-byte function codes mark commands which require more than one byte. They mark both the beginning and end of such functions in the file. The length is indicated for most functions, but some have a variable length. In these cases, programs should scan for the second occurrence of the function code to indicate the end of the function.

Length	Code	Command	Format	
6	$C0	Margin Reset	Byte:	$C0
			Byte:	Old left margin
			Byte:	Old right margin
			Byte:	New left margin
			Byte:	New right margin
			Byte:	$C0
4	$C1	Spacing reset	Byte:	$C1
			Byte:	Old spacing
			Byte:	New spacing
			Byte:	$C1
Note: Spacing values are stored in half-line increments.				
3	$C2	Left margin release	Byte:	$C2
			Byte:	Number of spaces to go left
			Byte:	$C2
5	$C3	Center the following text	Byte:	$C3
			Byte:	Type of center: 0 = between margins 1 = around current column
			Byte:	Center column number
			Byte:	Starting column number
			Byte:	$C3

			Bytes:	Centered text
			Byte:	$83
5	$C4	Align or flush right	Byte:	$C4
			Byte:	Align character (see below)
			Byte:	Align column number
			Byte:	Starting column number
			Byte:	$C4
			Bytes:	Aligned text
			Byte:	$84

Note: If the alignment character is $0A (hard new line) then this is a flush right and the alignment column number is the right margin; otherwise, the alignment column number is the next tab stop.

6	$C5	Reset hyphenation zone (hotzone)	Byte:	$C5
			Byte:	Old left hotzone
			Byte:	Old right hotzone
			Byte:	New left hotzone
			Byte:	New right hotzone
			Byte:	$C5
4	$C6	Set page number position	Byte:	$C6
			Byte:	Old position code
			Byte:	New position code Codes: 0 = None 1 = Top Left 2 = Top Center 3 = Top Right 4 = Top Left and Right 5 = Bottom Left 6 = Bottom Center 7 = Bottom Right 8 = Bottom Left and Right
			Byte:	$C6
6	$C7	Set page number	Byte:	$C7
			Byte:	Old number,high byte
			Byte:	Old number,low byte
			Byte:	New number,high byte
			Byte:	New number,low byte
			Byte:	$C7

Note: The page numbers are words with the bytes in the "wrong" order.

8	$C8	Set page number column positions	Byte:	$C8
			Byte:	Old left position
			Byte:	Old center position
			Byte:	Old right position
			Byte:	New left position
			Byte:	New center position
			Byte:	New right position
			Byte:	$C8
42	$C9	Set tabs	Byte:	$C9
			20 Bytes:	Old tab table

			20 Bytes:	New tab table
			Byte:	$C9

Note: Each bit in the tab table represents one position, from bit 0 to bit 159.

3	$CA	Conditional end of page	Byte:	$CA
			Byte:	Number of single-spaced lines not to be broken
			Byte:	$CA
6	$CB	Set pitch and/or font	Byte:	$CB
			Byte:	Old pitch
			Byte:	Old font
			Byte:	New pitch
			Byte:	New font
			Byte:	$CB

Note: If the pitch is negative, then it is proportional.

4	$CC	Set temporary margin (indent)	Byte:	$CC
			Byte:	Old temporary margin
			Byte:	New temporary margin
			Byte:	$CC
3	$CD	End of temporary margin	Byte:	$CD
			Byte:	Temporary margin
			Byte:	$CD
4	$CE	Set top margin	Byte:	$CE
			Byte:	Old top margin
			Byte:	New top margin
			Byte:	$CE
3	$CF	Suppress page characteristics	Byte:	$CF
			Byte:	Suppress code: (Any combination valid)
				Bit: Meaning:
				0 all suppressed
				1 Page numbers suppressed.
				2 Page number moved to bottom.
				3 All headers suppressed
				4 Header A suppressed
				5 Header B suppressed
				6 Footer A suppressed
				7 Footer B suppressed
			Byte:	$CF
6	$D0	Set form length	Byte:	$D0
			Byte:	Old form length

			Byte:	Old number of text lines
			Byte:	New form length
			Byte:	New number of text lines
			Byte:	$D0
Variable	$D1	Header/footer	Byte:	$D1
			Byte:	Old def byte (see below)
			Byte:	Number of half-lines used by old header/footer
			2 bytes:	$FF
			Byte:	Left margin
			Byte:	Right margin
			Bytes:	ASCII Text
			Byte:	$FF
			Byte:	Number of half-lines used by new header/footer
			Byte:	New def byte (see below)
			Byte:	$D1

The format of the def byte is as follows:

Bits 0-1:	Type:	0 = Header A	Bits 2-7:	Occurrence:	0 = Never
		1 = Header B			1 = All pages
		2 = Footer A			2 = Odd pages
		3 = Footer B			4 = Even pages

Note: The low-order two bits of the old def byte (the old types) must be correct.

Variable	$D2	Footnote	Byte:	$D2
			Byte:	Footnote number
			Byte:	Number of half-lines
			Byte:	$FF
			Byte:	Left margin
			Byte:	Right margin
			Bytes:	ASCII Text
			Byte:	$D2

Note: WordPerfect versions 1.0 and 1.1 use this function code. Versions 2.0 and later use function code $E2 instead.

4	$D3	Set Footnote Number	Byte:	$D3
			Byte:	Old footnote number
			Byte:	New footnote number
			Byte:	$D3

Note: WordPerfect versions 1.0 and 1.1 use this function code. Versions 2.0 and later use function code $E4 instead.

4	$D4	(Used in other WordPerfect Corporation Products)

4	$D5	Set lines per inch	Byte:	$D5
			Byte:	Old LPI code
			Byte:	New LPI code
			Byte:	$D5

Note: Only 6 or 8 lines per inch is valid.

6	$D6	Set extended tabs	Byte:	$D6
			Byte:	Old starting position
			Byte:	Old increment
			Byte:	New starting position
			Byte:	New increment
			Byte:	$D6

Note: The starting column position must be at least 160.

Variable	$D7	(Used in other WordPerfect Corporation Products)		
4	$D8	Set alignment character	Byte:	$D8
			Byte:	Old alignment character
			Byte:	New alignment character
			Byte:	$D8
4	$D9	Set left margin release (number of columns to go left)	Byte:	$D9
			Byte:	Old number
			Byte:	New number
			Byte:	$D9
4	$DA	Set underline mode	Byte:	$DA
			Byte:	Old mode (see below)
			Byte:	New mode (see below)
			Byte:	$DA

The underline mode is defined as follows:
0 = Normal Underline
1 = Double Underline
2 = Single Underline Continuous
3 = Double Underline Continuous

4	$DB	Set sheet feeder bin number	Byte:	$DB
			Byte:	Old number
			Byte:	New number
			Byte:	$DB

Note: The number is zero based (bin #1 is stored as 0).

Variable	$DC	End of page function (WordPerfect inserts this and it changes with each version)	Byte:	$DC
			Byte:	Number of 1/2 lines at end of page, low 7 bits
			Byte:	Number of 1/2 lines at end of page, high 7 bits
			Byte:	Number of 1/2 lines used for footnotes
			Byte*:	Number of pages used for footnotes
			Byte*:	Number of footnotes on this page
			Byte:	CEOP Flag
			Byte:	Suppress code
			Byte:	$DC

Note: Bytes marked with an asterisk (*) are fields present only in WordPerfect 2.0 and later.

24	$DD	(Used in other WordPerfect Corporation Products)		
4	$DE	End of temporary margin	Byte:	$DE
			Byte:	Old left temporary margin
			Byte:	Old right temporary margin
			Byte:	$DE
Variable	$DF	Invisible characters (embedded printer command)	Byte:	$DF
			Bytes:	7-bit text
			Byte:	$DF
4	$E0	Temporary margin	Byte:	$E0
			Byte:	New right temporary margin
			Byte:	New left temporary margin
			Byte:	$E0
3	$E1	(Used in other WordPerfect Corporation Products)		
Variable	$E2	New footnote/endnote (WordPerfect 2.0 and later)	Byte:	$E2
			Byte:	Def byte (see below)
			Byte:	Value A (see below)
			Byte:	Value B (see below)
			Byte:	Value C (see below)
			Byte:	Value D (see below)
			Byte:	Old footnote length in 1/2 lines
			Byte:	Number of lines on page 1
			Byte:	Number of lines on page 2
			Byte:	Number of lines on page 3
			.	
			.	
			.	
			Byte:	Number of lines on page N
			Byte:	Number of pages
			Byte:	$FF
			Byte:	Left margin
			Byte:	Right margin
			Bytes:	ASCII Text
			Byte:	$E2

The Def Byte is defined as follows:

Def Bit 0: 0 = use numbers
1 = use characters

Def Bit 1: 0 = footnote
1 = endnote

If Def Bit 0 = 0, then Values A and B are the footnote or endnote number taken together (see below.) If Def Bit 0 = 1, then Value A is the number of characters and Value B is the character.

Note: Values A and B (when taken together) and Values C and D (always) are 14-bit numbers split into 7-bit bytes, high order byte first.

Note: For endnotes, there is just a null between Value D and the $FF byte.

150	$E3	Footnote information/ options	Byte: $E3 74 Bytes: Old Footnote values 74 Bytes: New Footnote values Byte: $E3

The footnote values are defined as follows:

Byte: Spacing in footnotes (in half-lines)

Byte: Spacing between footnotes (in half-lines)

Byte: Number of lines to keep together

Byte: Flag byte:

- Bit 0: 1 if numbering starts on each page
- Bits 1-2: (for footnotes)
  - 0 = Use numbers
  - 1 = Use characters
  - 2 = Use letters
- Bits 3-4: (for endnotes)
  - 0 = Use numbers
  - 1 = Use characters
  - 2 = Use letters
- Bits 5-6:
  - 0 = No line separator
  - 1 = 2" line
  - 2 = Line from left to right margin
  - 3 = 2" line and continued message
- Bit 7:
  - 0 = footnotes after text
  - 1 = footnotes at bottom of page

Byte: Number of characters used in place of footnote numbers

5 Bytes: Characters used in place of footnote numbers (null terminated if less than five)

Byte: Number of displayable characters in string for footnote text

			15 Bytes:	String for footnote text
			Byte:	Number of displayable characters in string for endnote text
			15 Bytes:	String for endnote text
			Byte:	Number of displayable characters in string for footnote note
			15 Bytes:	String for footnote note
			Byte:	Number of displayable characters in string for endnote note
			15 Bytes:	String for endnote note
6	$E4	New set footnote (WordPerfect 2.0 and later)	Byte:	$E4
			Byte:	Old number,high byte
			Byte:	Old number,low byte
			Byte:	New number,high byte
			Byte:	New number,low byte
			Byte:	$E4

Note: The new number is zero based (stored as new number minus one)
Note: Footnote numbers are 14-bit numbers split into 7-bit bytes, high order byte first.

23	$E5	(Used in other WordPerfect Corporation Products)
11	$E6	(Used in other WordPerfect Corporation Products)
3	$E7	(Used in other WordPerfect Corporation Products)
3	$E8	(Used in other WordPerfect Corporation Products)
Variable	$E9	(Used in other WordPerfect Corporation Products)
Variable	$EA	(Used in other WordPerfect Corporation Products)
32	$EB	(Used in other WordPerfect Corporation Products)
4	$EC	(Used in other WordPerfect Corporation Products)
Variable	$ED	(Used in other WordPerfect Corporation Products)
44	$EE	(Used in other WordPerfect Corporation Products)
18	$EF	(Used in other WordPerfect Corporation Products)
6	$F0	(Used in other WordPerfect Corporation Products)
106	$F1	(Used in other WordPerfect Corporation Products)
Variable	$F2	(Used in other WordPerfect Corporation Products)
100	$F3	(Used in other WordPerfect Corporation Products)

END OF FILE FTN.A0.0000

```
##########################################################################
### FILE: FTN.ABOUT.92.06
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

About Apple II File Type Notes **June 1992**

This Note accompanies each release of Apple II File Type Notes. This release includes new Notes for file types $5A, $D8, $E0 and $E2, revised Notes for file types $B3, $B5, $B6, $B7 and $C7, as well as an updated list of all currently assigned Apple II file types and auxiliary types.

We welcome your file formats, suggestions on existing Notes, and your requests for file type or auxiliary type assignments. Please contact us at:

Apple II File Type Notes
Developer Technical Support
Apple Computer, Inc.
20525 Mariani Avenue, M/S 75-3T
Cupertino, CA 95014
AppleLink: DEVSUPPORT
Internet: DEVSUPPORT@AppleLink.Apple.com

The universal sharing of file formats opens new dimensions to personal computing, so we want Apple II File Type Notes distributed as widely as possible. We send them to all Partners and Associates at no charge, and we also post them on AppleLink in the Developer Services bulletin board and other electronic sources, including the Apple FTP site (IP 130.43.2.3). You can also order them through Resource Central. As a Resource Central customer, you have access to the tools and documentation necessary to develop Apple II-compatible products. For more information about Resource Central, contact:

Resource Central, Inc.
P.O. Box 11250
Overland Park, KS 66207
(913) 469-6502
Fax: (913) 469-6507
AppleLink: A2.CENTRAL
Internet: A2.CENTRAL@AppleLink.Apple.com
GEnie: RC.ELLEN

We place no restrictions on copying the Notes, with the exception that you cannot resell them. You should note, however, that some of the file formats are the copyrighted property of the companies which own them. These formats are identified in the appropriate Notes, and you should treat them with respect to the applicable copyright laws.

This File Type Note batch was originally released in May 1992. Since that time, many of the contact addresses have changed and some typographical errors have been fixed. To note these changes, this document now bears the date June 1992. NO CONTENT OF ANY NOTES HAS CHANGED SINCE MAY 1992.

All file type and auxiliary type combinations not listed in this Note are RESERVED and must not be used by applications without assignment from Apple Computer, Inc.

Specifically, you may NOT do the following:

- You may NOT use auxiliary type $0000 for a given file type instead of asking for an assignment.
- You may NOT use an auxiliary type in file type $BF, justifying it with, "My program runs under GS/OS and creates documents."
- You may NOT pick your own auxiliary type in a given file type and use it without getting an assignment from Apple. You may request a specific auxiliary type if you desire, but you must be prepared for the instance in which your requested auxiliary type is not available.
- You may NOT redefine the auxiliary type of a file type to suit your own purposes. For example, you can't choose to use auxiliary type $ABCD in file type $04, since the auxiliary type of file type $04 (text file) is already defined to be the record length of a random-access text file.

You may use file types and auxiliary types which are not assigned to you if a complete definition of the contents of the file is published in File Type Notes or elsewhere. For example, you do not need to be assigned auxiliary types to use text files or binary files.

If you have any questions at all about file type and auxiliary type policies, assignments, or other specifics, do not hesitate to contact Developer Technical Support at the address listed in this Note.

Developer Technical Support requires four things from a developer before publishing a file format for your application in a File Type Note:

1. The file format itself, preferably in an ASCII text file.
2. Assurances that the product is shipping. We don't want to jump the gun by releasing a Note for an unannounced product or a product which is unavailable. The enclosed list of file type assignments includes only those products which we know are currently shipping.

   Note : If your product is shipping, but your file type is not listed, you need to contact DTS, since engineering uses this list to identify files in future Apple products, such as future releases of the Apple IIgs Finder. If this Note does not list your file type and auxiliary type assignments, engineering CANNOT include them in future products. The "short" names listed in this Note are used as the descriptors for such files, so you should contact DTS if a descriptor for one of your files is unsuitable.

   You MUST inform DTS when the program using your file type

assignments ships to be included in future versions of the system software, even if you do not wish the file format to be published.

3. Written permission to publish the file format. We don't want you to submit the format and then be surprised when we publish it.

4. Your company name and address, so we can refer readers to you for more information about your product.

DEFINITIONS

The following definitions apply to all Apple II File Type Notes and will not be repeated in each Note:

BOOLEAN	A binary indicator stored as a word unless otherwise indicated. If any bit of a boolean is set, the boolean is TRUE. If it is clear, it is FALSE.
BYTE	An 8-bit value.
DOUBLE LONG	A 64-bit value, composed of eight byte, four words or two long words. The most significant byte is stored last.
FIXED	A four-byte signed value where the least significant word represents a fractional part and the most significant word represents an integer part (i.e., the value 32767.0 would be stored as $00 $00 $FF $7F to represent the integer part of $7FFF (+32767) and the fractional part of $0000 (0)). The value 4.5 would be stored as $00 $80 $04 $00 to represent the integer part of $0004 and the fractional part of $8000. The value $8000 is represented as 1000000000000000 in binary. The bit immediately following the decimal point is set, which indicates the value of 2^-1, or one-half. The full binary expansion of 4.5 is 0000000000000100.1000000000000000, which indicates 2^2 + 2^-1 or 4 + 0.5, which is 4.5. The Apple IIgs Integer Math Tools contain routines to assist with Fixed arithmetic.
FLAG UNIT	Any storage unit (byte, word, long) treated as a series of flag bits rather than as a numeric value.
LONG	A 32-bit value, composed of four bytes or two words. The most significant byte is stored last. For example, $00E102A8 would be stored as $A8 $02 $E1 $00.
REVERSE	The 65xxx series microprocessors normally store values with the least significant byte (LSB) first, while other microprocessors may store values with the most significant byte (MSB) first. The designation Reverse (Rev.) indicates that values must be rearranged before using them (i.e., a Long value of $11223344 would be stored as $44 $33 $22 $11, but a Reverse Long value would be stored as $11 $22 $33 $44).
STRING	A Pascal-type string. It consists of a length byte followed by up to 255 bytes of ASCII data.
WORD	A 16-bit value, composed of two bytes. The most significant byte of the word is stored after the least significant byte. For example, $02FF would be stored $FF $02.

All bit definitions are given as bit numbers. Bit 0 is always the least significant bit. The most significant bit of a byte is bit 7; the most significant bit of a word is bit 15, etc.

Following is a current list of all file type and auxiliary type assignments. Assignments with a date indicate the release date of the File Type Note for that assignment, and all file types and auxiliary types which are not listed in this Note are reserved and should not be used.

Although Apple strongly recommends the use of file type descriptors, this document includes a list of three-letter abbreviations solely for developer convenience. These abbreviations are final as documented and will not be changed.

FILE TYPE ASSIGNMENTS May 1992

New ***
Revised *R*

File Type	Aux. Type	3Ltr Abv	File Type Description (File Type Owner)	Auxiliary Type Description	Date
$xx	$xxxx	abc	12345678901234567890123	12345678901234567890123	xx/xx
$00		NON	Unknown		03/90
$01		BAD	Bad blocks		03/90
$02		PCD	Pascal code		
$03		PTX	Pascal text		
$04		TXT	ASCII text	Random-access record-length	
$05		PDA	Pascal data		
$06		BIN	Binary	Load address in bank 0	
$07		FNT	Apple /// Font		
$08		FOT	Apple II or /// Graphics		05/89
$08	$4000		Packed Hi-Res Image	Image Format	11/88
$08	$4001		Packed Double Hi-Res Image	Image Format	11/88
$08	$8001		Printographer Packed HGR file	Image Format	
$08	$8002		Printographer Packed DHGR file	Image Format	
$08	$8003		Softdisk Hi-Res image	Application-Specific	
$08	$8004		Softdisk Double Hi-Res image	Application-Specific	
$09		BA3	Apple /// BASIC program		
$0A		DA3	Apple /// BASIC data		
$0B		WPF	Apple II or /// Word Processor		
$0B	$8001		Write This Way document	Application Specific	
$0B	$8002		Writing & Publishing document	Application Specific	
$0C		SOS	Apple /// SOS System		
$0F		DIR	Folder		
$10		RPD	Apple /// RPS data		
$11		RPI	Apple /// RPS index		
$12		AFD	Apple /// AppleFile discard		
$13		AFM	Apple /// AppleFile model		
$14		AFR	Apple /// AppleFile report format		
$15		SCL	Apple /// screen library		
$16		PFS	PFS document		
$16	$0001		PFS:File document	Program Specific	
$16	$0002		PFS:Write document	Program Specific	
$16	$0003		PFS:Graph document	Program Specific	

$16	$0004		PFS:Plan document	Program Specific	
$16	$0016		PFS internal data	Program Specific	
$19		ADB	AppleWorks Data Base	Upper-/lowercase in name	07/90
$1A		AWP	AppleWorks Word Processor	Upper-/lowercase in name	09/89
$1B		ASP	AppleWorks Spreadsheet	Upper-/lowercase in name	09/89
$20		TDM	Desktop Manager document		
$21			Instant Pascal source		
$22			UCSD Pascal Volume		
$29			Apple /// SOS Dictionary		
$2A		8SC	Apple II Source Code	Application Specific	
$2B		8OB	Apple II Object Code	Application Specific	
$2B	$8001		GBBS Pro object Code	Application Specific	
$2C		8IC	Apple II Interpreted Code	Application Specific	
$2C	$8003		APEX Program File	Application Specific	
$2D		8LD	Apple II Language Data	Application Specific	
$2E		P8C	ProDOS 8 code module	Application Specific	
$2E	$8001		Davex 8 Command	Application Specific	
$2E	$8002	PTP	Point-to-Point drivers	Application Specific	
$2E	$8003	PTP	Point-to-Point code	Application Specific	
$40		DIC	Dictionary file	Application Specific	
$41			OCR data	Application Specific	
$41	$8001		InWords OCR font table	Application Specific	
$42		FTD	File Type Names	Search order	07/89
$43			Peripheral data	Application-Specific	
$50		GWP	Apple IIgs Word Processor		
$50	$8003		Personal Journal document	Application-Specific	
$50	$8011		Softdisk issue text	Application-Specific	
$50	$5445		Teach document	Application Specific	03/90
$50	$8001		DeluxeWrite document	Application Specific	
$50	$8010		AppleWorks GS Word processor	Application Specific	09/90
$51		GSS	Apple IIgs Spreadsheet		
$51	$8010		AppleWorks GS Spreadsheet	Application Specific	
$52		GDB	Apple IIgs Data Base		
$52	$8001		GTv database	Application Specific	
$52	$8010		AppleWorks GS Data Base	Application Specific	
$52	$8011		AppleWorks GS DB Template	Application Specific	
$52	$8013		GSAS database	Application Specific	
$52	$8014		GSAS accounting journals	Application Specific	
$52	$8015		Address Manager document	Application Specific	
$52	$8016		Address Manager defaults	Application-Specific	
$52	$8017		Address Manager index	Application-Specific	
$53		DRW	Drawing		
$53	$8002		Graphic Disk Labeler document	Application Specific	03/90
$53	$8010		AppleWorks GS Graphics	Application Specific	
$54		GDP	Desktop Publishing		
$54	$8002		GraphicWriter document	Application Specific	
$54	$8003		Label It document	Application-Specific	
$54	$8010		AppleWorks GS Page Layout	Application Specific	
$54	$DD3E		Medley document	Application Specific	05/89
$55		HMD	Hypermedia	Application Specific	
$55	$0001		HyperCard IIgs stack	Application Specific	
$55	$8001		Tutor-Tech document	Application Specific	
$55	$8002		HyperStudio document	Application Specific	
$55	$8003		Nexus document	Application Specific	
$55	$8004		HyperSoft stack	Application-Specific	
$55	$8005		HyperSoft card	Application-Specific	

$55 $8006		HyperSoft external command	Application-Specific	
$56	EDU	Educational Data	Application Specific	
$56 $8001		Tutor-Tech Scores	Application Specific	
$56 $8007		GradeBook Data	Applcation-specific	
$57	STN	Stationery		
$57 $8003		Music Writer format	Application Specific	
$58	HLP	Help File		
$58 $8002		Davex 8 Help File	Application Specific	
$58 $8006		Locator help document	Application-Specific	
$58 $8007		Personal Journal help	Application-Specific	
$58 $8008		Home Refinancer help	Application-Specific	
$59	COM	Communications File	Application Specific	
$59 $8010		AppleWorks GS Communications	Application Specific	
$5A	CFG	Configuration file	Application Specific	***05/92
$5A $0000		Sound settings files	Identified by name	***05/92
$5A $0002		Battery RAM configuration		05/90
$5A $0003		AutoLaunch preferences	Application Specific	
$5A $0005		GSBug configuration	Application Specific	
$5A $8001		Master Tracks Jr. preferences		
$5A $8002		GraphicWriter preferences	Application Specific	
$5A $8003		Z-Link configuration	Application Specific	
$5A $8004		JumpStart configuration	Application Specific	
$5A $8005		Davex 8 configuration	Application Specific	
$5A $8006		Nifty List configuration	Application Specific	
$5A $8007		GTv videodisc configuration	Application Specific	
$5A $8008		GTv Workshop configuration	Applicatoin Specific	
$5A $8009	PTP	Point-to-Point preferences	Application Specific	
$5A $800A		ORCA/Disassembler preferences	Application Specific	
$5A $800B		SnowTerm preferences	Application Specific	
$5A $800C		My Word! preferences	Application Specific	
$5A $800D		Chipmunk configuration	Application Specific	
$5A $8010		AppleWorks GS configuration	Application Specific	
$5A $8011		SDE Shell preferences	Application Specific	
$5A $8012		SDE Editor preferences	Application Specific	
$5A $8013		SDE system tab ruler	Application Specific	
$5A $8014		Nexus configuration	Application Specific	
$5A $8015		DesignMaster preferences	Application Specific	
$5A $801A		MAX/Edit keyboard template	Application-Specific	
$5A $801B		MAX/Edit tab ruler set	Application-Specific	
$5A $801C		Platinum Paint preferences	Application Specific	
$5A $801D		Sea Scan 1000 preferences	Application-Specific	
$5A $801E		Allison preferences	Application Specific	
$5A $801F		Gold of the Americas options	Application-Specific	
$5A $8021		GSAS accounting setup	Application Specific	
$5A $8023		UtilityLaunch preferences	Application-Specific	
$5A $8024		Softdisk configuration	Application Specific	

$5A $8025		Quit-To configuration	Application Specific	
$5A $8026		Big Edit Thing preferences	Application-Specific	
$5A $8027		ZMaker preferences	Application-Specific	
$5A $8028		Minstrel configuration	Application-Specific	
$5A $8029		WordWorks Pro preferences	Application-Specific	
$5A $802B		Pointless preferences	Application-Specific	
$5A $802E		Label It configuration	Application-Specific	
$5A $802F		Cool Cursor document	Application-Specific	***05/92
$5A $8030		Locator preferences	Application-Specific	
$5A $8031		Replicator preferences	Application-Specific	***05/92
$5A $8035		Home Refinancer preferences	Application-Specific	
$5B	ANM	Animation file	Application Specific	
$5B $8001		Cartooners movie	Application Specific	
$5B $8002		Cartooners actors	Application Specific	
$5B $8005		Arcade King Super document	Application Specific	
$5B $8006		Arcade King DHRG document	Application Specific	
$5B $8007		DreamVision movie	Application Specific	
$5C	MUM	Multimedia document	Application Specific	
$5C $8001		GTv multimedia playlist	Application Specific	
$5D	ENT	Game/Entertainment document	Application Specific	
$5D $8001		Solitaire Royale document	Application Specific	
$5D $8002		BattleFront scenario		
$5D $8003		BattleFront saved game		
$5D $8004		Gold of the Americas game	Application-Specific	
$5D $8006		Blackjack Tutor document	Application Specific	
$5D $8010		Quizzical high scores	Application-Specific	
$5D $8011		Meltdown high scores	Application-Specific	
$5D $8012		BlockWords high scores	Application-Specific	
$5E	DVU	Development utility document	Application Specific	
$5E $0001		Resource file		05/90
$5E $8001		ORCA/Disassembler template	Application Specific	
$5E $8003		DesignMaster document	Application Specific	
$5F	FIN	Financial document	Application Specific	
$5F $8002		Home Refinancer document	Application-Specific	
$6B	BIO	PC Transporter BIOS		
$6D	TDR	PC Transporter driver		
$6E	PRE	PC Transporter pre-boot		
$6F	HDV	PC Transporter volume		
$A0	WP	WordPerfect document	WordPerfect	01/89
$AB	GSB	Apple IIgs BASIC program		
$AC	TDF	Apple IIgs BASIC TDF		
$AD	BDF	Apple IIgs BASIC data		
$B0	SRC	Apple IIgs source code	APW Language type	07/90
$B0 $0001		APW Text file		
$B0 $0003		APW 65816 Assembly source code		
$B0 $0005		ORCA/Pascal source code		
$B0 $0006		APW command file		
$B0 $0008		ORCA/C source code		
$B0 $0009		APW Linker command file		
$B0 $000A		APW C source code		
$B0 $000C		ORCA/Desktop command file		
$B0 $0015		APW Rez source file		

$B0	$0017		Installer script	Application-Specific	
$B0	$001E		TML Pascal source code		
$B0	$0116		ORCA/Disassembler script	Application Specific	
$B0	$0503		SDE Assembler source code	Application Specific	
$B0	$0506		SDE command script	Application Specific	
$B0	$0601		Nifty List data	Application-Specific	
$B0	$0719		PostScript file	Application-Specific	
$B1		OBJ	Apple IIgs object code		
$B2		LIB	Apple IIgs Library file		
$B3		S16	GS/OS application		*R*05/92
$B4		RTL	GS/OS Run-Time Library		
$B5		EXE	GS/OS Shell application		*R*05/92
$B6		PIF	Permanent initialization file	Not loaded if bit 15 set	*R*05/92
$B7		TIF	Temporary initialization file	Not loaded if bit 15 set	*R*05/92
$B8		NDA	New desk accessory	Not loaded if bit 15 set	09/90
$B9		CDA	Classic desk accessory	Not loaded if bit 15 set	09/90
$BA		TOL	Tool		09/90
$BB		DVR	Apple IIgs Device Driver file	Not loaded if bit 15 set	11/89
$BB	$7F01		GTv videodisc serial driver	Application Specific	
$BB	$7F02		GTv videodisc game port driver	Application Specific	
$BC		LDF	Load file (generic)		07/90
$BC	$4001		Nifty List Module	Application Specific	
$BC	$4002		Super Info module	Application Specific	03/91
$BC	$4004		Twilight document	Application Specific	
$BC	$4007		HyperStudio New Button Action	Application-Specific	
$BC	$4008		HyperStudio Screen Transition	Application-Specific	
$BC	$4009		DreamGrafix module	Application Specific	
$BC	$400A		HyperStudio Extra utility	Application-Specific	
$BD		FST	GS/OS File System Translator	Not loaded if bit 15 set	09/90
$BF		DOC	GS/OS document		
$C0		PNT	Packed Super Hi-Res picture	Application Specific	
$C0	$0000		Paintworks Packed picture	Application Specific	11/88
$C0	$0001		Packed Super Hi-Res Image	Application Specific	11/88
$C0	$0002		Apple Preferred Format picture	Application Specific	12/91
$C0	$0003		Packed QuickDraw II PICT file	Application Specific	11/88
$C0	$8001		GTv background image	Application Specific	
$C0	$8005		DreamGrafix document	Application Specific	
$C0	$8006		GIF document	Application-Specific	
$C1		PIC	Super Hi-Res picture	Application Specific	11/88
$C1	$0000		Super Hi-Res Screen image	Application Specific	11/88
$C1	$0001		QuickDraw PICT file	Application Specific	11/88
$C1	$0002		Super Hi-Res 3200 color screen image		09/90
$C1	$8001		Allison raw image doc.	Application Specific	
$C1	$8002		ThunderScan image doc.	Application Specific	
$C1	$8003		DreamGrafix document	Application-Specific	
$C2		ANI	Paintworks animation		
$C3		PAL	Paintworks palette		

$C5		OOG	Object-oriented graphics	Application Specific	
$C5	$8000		Draw Plus document	Application Specific	
$C5	$C000		DYOH: Architecture doc.	Application Specific	
$C5	$C001		DYOH predrawn objects	Application Specific	
$C5	$C002		DYOH custom objects	Application Specific	
$C5	$C003		DYOH clipboard	Application Specific	
$C5	$C004			Application Specific	
$C5	$C005			Application Specific	
$C5	$C006		DYOH: Landscape Document	Application Specific	
$C5	$C007		PyWare Document	Application-Specific	
$C6		SCR	Script	Application Specific	
$C6	$8001			Application Specific	
$C7		CDV	Control Panel document	Not loaded if bit 15 set	*R*05/92
$C8		FON	Font		
$C8	$0000		Font (Standard Apple IIgs QuickDraw II Font)	Reserved	01/89
$C8	$0001		TrueType font	Application-Specific	
$C9		FND	Finder data		
$CA		ICN	Icons		07/89
$D5		MUS	Music sequence	Application Specific	01/90
$D5	$0000		Music Construction Set song	Application Specific	
$D5	$0001		MIDI Synth sequence		
$D5	$0007		SoundSmith document	Application Specific	03/90
$D5	$8002		Diversi-Tune sequence	Application Specific	
$D5	$8003		Master Tracks Jr. sequence	Application Specific	
$D5	$8005		Arcade King Super music	Application Specific	
$D6		INS	Instrument	Application Specific	01/90
$D6	$0001		MIDI Synth instrument		
$D6	$0000		Music Construction Set instrument	Application Specific	
$D6	$8002		Diversi-Tune instrument	Application Specific	
$D7		MDI	MIDI data		01/90
$D7	$0000		MIDI standard data	Application Specific	
$D8		SND	Sampled sound	Application Specific	01/90
$D8	$0000		Audio IFF document	Application Specific	03/91
$D8	$0001		AIFF-C document	Application Specific	03/91
$D8	$0002		ASIF instrument	Application Specific	03/89
$D8	$0003		Sound resource file	Application-Specific	***05/92
$D8	$0004		MIDI Synth wave data		
$D8	$8001		HyperStudio sound	Application Specific	05/90
$D8	$8002		Arcade King Super sound	Application Specific	
$D8	$8003		SoundOff! sound bank	Application-Specific	
$DB		DBM	DB Master document	Application Specific	
$DB	$0001		DB Master document	Application Specific	
$E0		LBR	Archival library	Application Specific	
$E0	$0000		ALU library	Carolina System Software	
$E0	$0001		AppleSingle File	Application Specific	11/90
$E0	$0002		AppleDouble Header File	Application Specific	11/90
$E0	$0003		AppleDouble Data File	Application Specific	11/90
$E0	$0005		DiskCopy disk image	Application-Specific	***05/92
$E0	$8000		Binary II File	Application Specific	07/89
$E0	$8001		AppleLink ACU document	Application Specific	
$E0	$8002		ShrinkIt (NuFX) document	Application Specific	07/90
$E0	$8004		Davex archived volume	Application Specific	05/90
$E0	$8006		EZ Backup Saveset doc.	Application Specific	09/90
$E0	$8007		ELS DOS 3.3 volume	Application Specific	

```
$E0 $8009       UtilityWorks document     Application-Specific
$E0 $800A       Replicator document       Application-Specific       ***05/92
$E2       ATK   AppleTalk data
$E2 $FFFF       EasyMount document                                   ***05/92
$EE       R16   EDASM 816 relocatable file
$EF       PAS   Pascal area
$F0       CMD   BASIC command
$F1             User #1
$F2             User #2
$F3             User #3
$F4             User #4
$F5             User #5
$F6             User #6
$F7             User #7
$F8             User #8
$F9       OS    GS/OS System file
$FA       INT  Integer BASIC program
$FB       IVR  Integer BASIC variables
$FC       BAS  AppleSoft BASIC program
$FD       VAR  AppleSoft BASIC variables
$FE       REL  Relocatable code
$FF       SYS  ProDOS 8 application


### END OF FILE FTN.ABOUT.92.06
```

```
########################################################################
### FILE: FTN.B0.xxxx
########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: **$B0 (176)**
Auxiliary Type: **All**

Full Name: **Apple IIgs Source Code File**
Short Name: **Apple IIgs Source Code**

Revised by: **Matt Deatherage** **July 1990**
Written by: **Jim Merritt & Matt Deatherage** **May 1990**

Files of this type contain source code to be compiled by Apple IIgs native development environments. This File Type Note obsoletes any publications bearing this information with earlier publication dates.
Changes since March 1990: Added new vendor and language numbers and altered the warning about resource forks.

Files of type $B0 contain source code to be used by an Apple IIgs native development environment, such as APW. The data fork of $B0 files contains straight ASCII text; nothing that should not be interpreted as ASCII should be in this kind of file. The resource fork is reserved; it cannot be guaranteed that resources in these files are preserved by programming utilities. The resource fork, if it exists, should contain Apple IIgs resources.

The auxiliary type of file type $B0 is used to distinguish between different compilers or assemblers. The value in the auxiliary type is also referred to as the "APW Language Number," since APW matches language numbers with the auxiliary type of $B0 files to determine which compiler or assembler should be used to compile a source code file. Throughout this Note, "APW Language Number" is used to mean "auxiliary type of file type $B0," even though a specific auxiliary type value may be assigned to a native development environment other than APW, and may, in fact, not represent an APW language.

Developer Technical Support assigns and catalogs all official APW language numbers, and effective May 1988, DTS has a new scheme for these numbers. Under the new scheme, the high byte of the APW language number is a vendor number and the low byte is a language number. To form the APW language number, combine the vendor number with the language number.

Note: Vendors who form their own language numbers in this fashion should also inform Developer Technical Support of the new products to ensure proper inclusion in future File Type Descriptors.

Following is a list of the APW languages which do not follow the new scheme; inclusion of a language on this list does not imply the language product exists or ever will exist under APW.

Number	Language Code	Use

$0	PRODOS	Text file (File Type $04)
$1	Text	APW text file
$2	ASM6502	6502 Assembler
$3	ASM65816	65816 Assembler
$4	BASIC	Byte Works BASIC
$5	BWPASCAL	Byte Works Pascal
$6	EXEC	Command file
$7	SMALLC	Byte Works small C
$8	BWC	Byte Works C
$9	LINKED	APW linker command language
$A	CC	APW C
$B	PASCAL	APW Pascal
$C	COMMAND	Byte Works command-processor window
$1E	TMLPASCAL	TML Pascal

The following is a list of currently defined vendors and languages; inclusion of a vendor on this list does not imply the vendor is developing, or ever will be developing, any of the language products listed for APW.

Vendor Number	Vendor Name
$0	Apple Computer
$1	The Byte Works
$2	TML Systems
$3	Zedcor
$4	RavenWare
$5	SEA Software
$6	DAL Systems
$7	Adobe Systems

Language Number	Language Name
$2	6502 Assembler
$3	65816 Assembler
$4	BASIC
$6	Script files
$9	Linker command file
$A	C
$B	Pascal
$C	Command-processor window
$D	Forth
$E	Small C
$F	Lisp
$10	Modula-2
$11	FORTRAN
$12	Logo
$13	Prolog
$14	COBOL
$15	Resource Description
$16	Disassembly template
$17-$18	Reserved
$19	Page description

The generic vendor names and language descriptions are more familiar in

combination. For example, an "Apple Resource Description" file would be Rez source code, and "Adobe Systems Page Description" is a Postscript(TM) file.

If, as a developer of native development software, you need a vendor number or a new language number for a language processor not currently covered on this list, write to the address in "About File Type Notes", to the attention of "APW Language Number Administration".

Note: Language number assignments are considered provisional until the applicant submits proof of publication of a language processor using the assigned number. Acceptable proof must include a complete specification for the language that the processor recognizes, as well as photocopies of public notices that discuss the terms and details of publication (e.g., newspaper and magazine ads, software reviews, brochures, circulars, electronic mail solicitations, etc.). Unless a developer has made prior arrangements with Developer Technical Support, DTS may rescind a provisional language number assignment after a period of one calendar year from the date of assignment if a developer does not submit the required proof of publication.

END OF FILE FTN.B0.xxxx

```
##########################################################################
### FILE: FTN.B3.XXXX
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $B3 (179)
Auxiliary Type: All

Full Name: ProDOS 16 or GS/OS Application
Short Name: GS/OS Application

Revised by: Dave Lyons May 1992
Written by: Dave Lyons & Matt Deatherage September 1989

Files of this type and auxiliary type contain application programs for the Apple IIgs.

CHANGES SINCE DECEMBER 1991: Broadened the definition auxiliary type bit 1 to mean the application can handle getting control with the Super Hi-Res screen on.

Files of type $B3 contain GS/OS application programs. These files contain program code in Object Module Format (OMF) that is loaded by the System Loader or ExpressLoad at an address and is then executed in a documented environment.

The environment and constraints for application programs are documented in GS/OS Reference; all developers creating file of type $B3 should be familiar with this material. OMF is documented in _GS/OS Reference_ and the _APW Reference Manual._

The auxiliary type for $B3 files is now defined to indicate properties of the program contained within the file. Other parts of the system may use this information to properly control the environment for the program:

bits 31-16	reserved--must be 0
bits 15-8	signature byte. $DB means bits 7-0 are valid
bits 7-3	reserved--must be 0
bit 2	Message Aware: 1 = uses Message Center message #1 0 = ignores Message Center message #1
bit 1	Desktop Application: 1 = application can handle the Super Hi-Res screen already being on when it first gets control, so the system can provide a smooth visual transition into the application 0 = application is not prepared for the Super Hi-Res screen to be on
bit 0	GS/OS Aware: 1 = uses long prefixes (for example, prefix 9 instead of prefix 1)

0 = uses short prefixes (less than 63 characters)

NOTE: If an application has the Desktop Application bit set, it should be prepared to get control with either the text or the Super Hi-Res screen visible. For example, if some error prevents the application from using the desktop tools, it may be necessary to call GrafOff before the user can read error messages displayed on the text screen (although GrafOff is a QuickDraw II call, it's OK to call GrafOff even if QuickDraw II is not active).

If an application does not have the Desktop Application bit set (or does not even have a $DBxx auxiliary type), the system software reserves the right to force the text screen to be visible if QuickDraw II is not started. Do not assume that a Quit call from one application to another (with QuickDraw II not started) will leave the Super Hi-Res screen visible.

END OF FILE FTN.B3.XXXX

```
##########################################################################
### FILE: FTN.B5.XXXX
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $B5 (181)
Auxiliary Type: All

Full Name: ProDOS 16 or GS/OS Shell application file
Short Name: GS/OS Shell application

Revised by: Dave Lyons — May 1992
Written by: Dave Lyons & Matt Deatherage — September 1989

Files of this type and auxiliary type contain application programs intended for use within a shell environment for the Apple IIgs.

CHANGES SINCE DECEMBER 1991: Broadened the definition auxiliary type bit 1 to mean the application can handle getting control with the Super Hi-Res screen on.

Files of type $B5 contain GS/OS shell application programs. These files contain program code in Object Module Format (OMF) that is loaded by the System Loader or ExpressLoad at an address and is then executed under the control of a command shell (such as APW, for example).

The shell may provide extra services to a shell application that are not available to normal GS/OS applications (files of type $B3). A shell application can identify which shell it is running under by examining the shell identifier.

Information about the shell identifier and other shell application environment issues may be found in GS/OS Reference and APW Reference, where the shell application environment is completely documented. OMF is documented in those manuals as well. All developers creating files of type $B5 should be familiar with this material.

The auxiliary type for $B5 files is now defined to indicate properties of the program contained within the file. Other parts of the system may use this information to properly control the environment for the program:

bits 31-16	reserved--must be 0
bits 15-8	signature byte. $DB means bits 7-0 are valid
bits 7-3	reserved--must be 0
bit 2	Message Aware: 1 = uses Message Center message #1 0 = ignores Message Center message #1
bit 1	Desktop Application: 1 = application can handle the Super Hi-Res screen already being on when it first gets control, so the system can provide a smooth visual transition into the application

```
                    0 = application is not prepared for the Super
                        Hi-Res screen to be on
      bit 0         GS/OS Aware:
                    1 = uses long prefixes (for example, prefix 9
                        instead of prefix 1)
                    0 = uses short prefixes (less than 63
                        characters)

   NOTE: If a Shell Application has the Desktop Application bit
         set, it should be prepared to get control with either the
         text or the Super Hi-Res screen visible.  For example, if
         some error prevents the application from using the
         desktop tools, it may be necessary to make the QuickDraw
         II call GrafOff before the user can read error messages
         displayed on the text screen (it's OK to call GrafOff
         even if QuickDraw II is not active).

         If a Shell Application does not have the Desktop
         Application bit set (or does not even have a $DBxx
         auxiliary type), the system software reserves the right
         to force the text screen to be visible if QuickDraw II is
         not started.  Do not assume that a Quit call from one
         application to another (with QuickDraw II not started)
         will leave the Super Hi-Res screen visible.

### END OF FILE FTN.B5.XXXX
```

```
##########################################################################
### FILE: FTN.B6.XXXX
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: **$B6 (182)**
Auxiliary Type: **All**

Full Name: **ProDOS 16 or GS/OS Permanent Initialization File**
Short Name: **Permanent initialization file**

Modified by: **Matt Deatherage** **May 1992**
Written by: **Matt Deatherage** **September 1990**

Files of this type contain initialization code that does not get unloaded.

CHANGES SINCE SEPTEMBER 1990: Added new information pertaining to System Software 6.0 and answered some commonly asked questions.

Files of type $B6 contain permanent initialization code in OMF format. Such files are often referred to as "inits." They are loaded by GS/OS at boot time and are never unloaded. The auxiliary type is RESERVED except for bit 15--if bit 15 is set, the initialization file is not loaded at boot time.

The structure of an init is similar to that of an application. The first byte of the loaded code image (inits are load files) is the entry point, and the init must end with an RTL instruction. When GS/OS transfers control to a permanent initialization file, the processor is in 16-bit native mode. The A register contains the init's user ID, D points to the bottom of a 4K stack and direct-page area and S points to near the top of that area. (If the init has an OMF stack and direct page segment linked in, the D and S registers point to it instead.) The data bank register is not defined; you should save it, set it and restore it if you use absolute addressing.

While all tools are available to be started, that doesn't mean tools should necessarily be started. Inits can be loaded after boot time (such as with IR 2.0, DTS Sample Code for an Apple IIgs Finder Extension), and blindly attempting to start and shut down tools without first checking their status can be disastrous in such instances. In particular, inits should never call TLStartUp or TLShutDown, and should check for the presence of other tools through each tool's status function before starting it up.

If you're considering starting a tool after init time and leaving it started (which is only possible when your code gets control after init time), you must do two things:

1. **Apply for your Certificate of IIgs Deityship at the Matt & Dave Ministry of Bits. Do not continue until you recieve your certificate.**
2. **Read Apple IIgs Technical Note #53, "NDAs and Tools," and do what it says.**

Inits that need to tell the difference between boot time and later loading times (for example, a RAM disk restoration init) can check the result of the GS/OS call GetName--if there is no name, the system's currently being started up.

Permanent inits are called at boot time and left in memory until the system is shut down. However, GS/OS does not call them again (even on a return from ProDOS 8). If your permanent init wants to periodically get control, it can use features like heartbeat tasks (installed with SetHeartBeat and IntSource), GS/OS notification procedures (AddNotifyProc), inter-process communication features (AcceptRequests) or Run Queue tasks (AddToRunQ).

Your permanent init can tell GS/OS it should be unloaded after execution. Above GS/OS's RTL address on the stack is a WORD value of $0000. If your init sets bit zero of this word (LDA #1, STA 4,S assuming you haven't pushed anything on the stack), GS/OS unloads your init when you return control, treating it as if it were a temporary init file. This can be useful for inits that operate with certain hardware--if the hardware isn't present, the init can go away.

WARNING: This WORD space is not available to permanent initialization files that execute from a user's folder on an AppleShare file server at boot time unless you're using System Software 6.0 or later.

Further Reference

- GS/OS Reference
- File Type Note for File Type $B7, Temporary Initialization Files

END OF FILE FTN.B6.XXXX

```
##########################################################################
### FILE: FTN.B7.XXXX
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: **$B7 (183)**
Auxiliary Type: **All**

Full Name: **ProDOS 16 or GS/OS Temporary Initialization File**
Short Name: **Temporary initialization file**

Modified by: **Matt Deatherage** **May 1992**
Written by: **Matt Deatherage** **September 1990**

Files of this type contain initialization code that is unloaded immediately after executing.

CHANGES SINCE SEPTEMBER 1990: Added new information pertaining to System Software 6.0 and answered some commonly asked questions.

Files of type $B7 contain temporary initialization code. Such files are often referred to as "inits". They are loaded by GS/OS at boot time and are unloaded immediately after execution. The auxiliary type is RESERVED except for bit 15--if bit 15 is set, the initialization file is not loaded.

The structure of an init is similar to that of an application. The first byte of the loaded code image (inits are load files) is the entry point, and the init must end with an RTL instruction. When GS/OS transfers control to a temporary initialization file, the processor is in 16-bit native mode. The A register contains the init's user ID, D points to the bottom of a 4K stack and direct-page area and S points to near the top of that area. (If the init has an OMF stack and direct page segment linked in, the D and S registers point to it instead.) The data bank register is not defined; you should save it, set it and restore it if you use absolute addressing.

All inits are loaded and executed entirely after the System Software is initialized; all of GS/OS is present and all of the tools are startable (although that's not necessarily advised; see later in this Note). The contents of all prefixes are undefined, and you should save and restore any prefixes you use. An init that wants to find its own pathname can use the System Loader call LGetPathname2. The commonly-seen icons at the bottom of the graphics screen are only available to CDevs in System Software 5.0 through 5.0.4, unless you draw the icon yourself, but under 6.0 and later the Miscellaneous Tools call ShowBootInfo will display an icon on the graphics screen or a version line on the text screen like GS/OS components.

While all tools are available to be started, that doesn't mean tools should necessarily be started. Inits can be loaded after boot time (such as with IR 2.0, DTS Sample Code for an Apple IIgs Finder Extension), and blindly attempting to start and shut down tools without first checking their status can be disastrous in such instances. In particular, inits should never call TLStartUp or TLShutDown, and should check for the presence of other tools

through each tool's status function before starting it up.

Inits that need to tell the difference between boot time and later loading times (for example, a RAM disk restoration init) can check the result of the GS/OS call GetName--if there is no name, the system's currently being started up.

Temporary initialization files are shut down by GS/OS after they perform their RTL, so they are a good choice for transient purposes. Temporary inits are good for playing sounds during the boot process, loading pictures, and other instances where data is passed to other system routines. For example, a temporary init might read files from a disk and save them to a RAM disk. The init gets to set up the RAM disk, but after that's done it doesn't need to stick around and take up memory--and since it's a temporary init, GS/OS unloads it after its work is done.

Further Reference

- GS/OS Reference
- File Type Note for File Type $B6, Permanent Initialization Files

END OF FILE FTN.B7.XXXX

```
##########################################################################
### FILE: FTN.B8.xxxx
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $B8 (184)
Auxiliary Type: All

Full Name: Apple IIgs New Desk Accessory
Short Name: New Desk Accessory

Written by: Dave "Of course I trust you, I'm just checking" Lyons — September 1990

Files of this type contain Apple IIgs New Desk Accessories in OMF format.

New Desk Accessories are available under the Apple menu of Apple IIgs desktop applications.

Files of type $B8 in the System:Desk.Accs directory of the boot volume are automatically loaded with the Loader and installed into the system with InstallNDA at boot time. Bit 15 of the auxiliary type is the "inactive" bit--if it's set, the system does not load the file. All other bits in the auxiliary type are reserved and should be zero.

NDAs are stored on disk in Object Module Format, which is defined in Appendix F of GS/OS Reference for GS/OS System Software 5.0. Once the Loader brings your NDA's code into memory, the format is as specified in the Desk Manager chapter of Apple IIgs Toolbox Reference, volume 1.

NDA authors should be sure to read the material listed under "Further Reference."

Further Reference

- Apple IIgs Toolbox Reference, Volumes 1-3
- GS/OS Reference for System Software 5.0
- Apple IIgs Technical Note #12, Tool Set Interdependencies
- Apple IIgs Technical Note #53, Desk Accessories and Tools
- Apple IIgs Technical Note #71, Desk Accessory Tips and Techniques
- Apple IIgs Technical Note #83, Resource Manager Stuff
- Apple IIgs Technical Note #84, TaskMaster Madness
- Apple IIgs Sample Code Note #19, ActionNDA

```
### END OF FILE FTN.B8.xxxx
```

```
##########################################################################
### FILE: FTN.B9.xxxx
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $B9 (185)
Auxiliary Type: All

Full Name: Apple IIgs Classic Desk Accessory
Short Name: Classic Desk Accessory

Written by: Dave "Mr. Nifty" Lyons September 1990

Files of this type contain Apple IIgs Classic Desk Accessories in OMF format.

Classic Desk Accessories are available by pressing Command-Ctrl-Esc, whether or not a desktop application is running.

Files of type $B9 in the System:Desk.Accs directory of the boot volume are automatically loaded with the Loader and installed into the system with InstallCDA at boot time. Bit 15 of the auxiliary type is the "inactive" bit--if it's set, GS/OS does not load the CDA. All other bits in the auxiliary type are reserved and should be zero.

CDAs are stored on disk in Object Module Format, which is defined in Appendix F of GS/OS Reference. Once the Loader brings your CDA's code into memory, the format is as specified in the Desk Manager chapter of Apple IIgs Toolbox Reference, volume 1.

CDA authors should be sure to read the material listed under "Further Reference."

Further Reference

- Apple IIgs Toolbox Reference, Volumes 1 and 3
- GS/OS Reference
- Apple IIgs Technical Note #71, Desk Accessory Tips and Techniques

END OF FILE FTN.B9.xxxx

```
##########################################################################
### FILE: FTN.BA.xxxx
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $BA (186)
Auxiliary Type: All

Full Name: Apple IIgs Tool File
Short Name: Tool

Written by: Matt Deatherage September 1990

Files of this type and auxiliary type contain Apple IIgs tool sets.

Files of type $BA contain Apple IIgs tool sets, designed to be called by the Tool Locator. The data fork is in OMF format; once it's been loaded by InitialLoad or InitialLoad2, the image in RAM starts with the tool set's function pointer table (FPT). The resource fork, if it exists, should be empty or in Apple IIgs Resource Manager format.

All RAM-based system tools are contained in the System folder of the boot volume, and are named by tool number (for example, tool #27 is named TOOL027). This convention is for system tools only.

User tools can reside in any directory (the directory with the application is a good choice) and may follow any naming conventions. User tool sets should also use file type $BA.

Please note that all system tool set numbers and names are reserved for Apple Computer, Inc. Third-party developers must not create tools using any system tool set numbers.

Further Reference

- Apple IIgs Toolbox Reference, Volume 2, Appendix A
- Apple IIgs Technical Note #73, Using User Tool Sets

END OF FILE FTN.BA.xxxx

```
##########################################################################
### FILE: FTN.BB.xxxx
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: **$BB (187)**
Auxiliary Type: **All**

Full Name: **Apple IIGS Device Driver File**
Short Name: **Device driver**

Written by: **Matt Deatherage** **November 1989**

Files of this type and all auxiliary types contain Apple IIGS device drivers.

An Apple IIGS device driver is defined as code that is necessary to control or operate a peripheral in a desired manner. A driver file's data fork contains code (in Object Module Format) that can be loaded by the System Loader or ExpressLoad and executed.

The format of a driver file depends largely upon the auxiliary type of the driver. Each driver has a class and an inactive flag in the auxiliary type, as Figure 1 indicates.

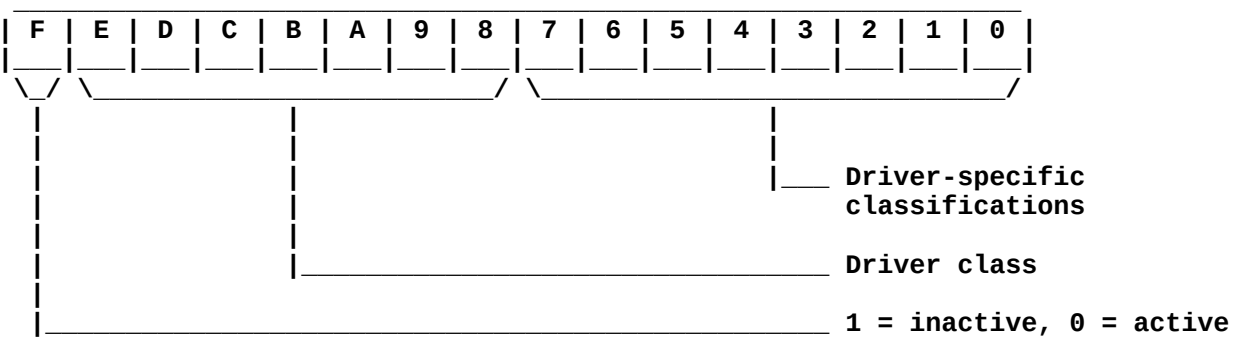


Figure 1-Auxiliary Type Definition

If bit 15 of the auxiliary type is set, the driver is considered inactive and should not be used. If the bit is clear, the driver is considered active and may be used if desired. Users may inactivate drivers from the Finder and are likely to be confused if a driver they marked as inactive continues to appear.

Bits 14 through 8 determine the driver's class. This mechanism allows many types of device drivers to share the same OMF-compatible file type. Table 1 lists defined classes of drivers.

Class	Definition
$00	Print Manager driver

$01	GS/OS driver
$02	AppleTalk driver
$03	MIDI Tools driver
$04--$7E	Reserved
$7F	Third-party multimedia driver

Table 1-Driver Classes

The low byte of the auxiliary type (bits 7 through 0) is referred to as the subclass, and it depends upon the driver class for interpretation. Below are the interpretations for the defined driver classes.

Print Manager Drivers

For class $00, the low byte determines the kind of Print Manager driver is contained in the file. A subclass of $00 indicates a printer driver, a subclass of $01 indicates a directly-connected port driver, and a subclass of $02 indicates a network port driver. All other values in the subclass for Print Manager drivers are reserved.

Printer and Port Drivers are documented in Apple IIGS Technical Note #35, Printer Driver Specifications and in Apple IIGS Technical Note #36, Port Driver Specifications.

GS/OS Drivers

GS/OS drivers are class $01. GS/OS groups the subclass into two fields. Bits 7 and 6 indicate the GS/OS driver type. A value of 00 indicates a standard GS/OS device driver. A value of 01 indicates a GS/OS Supervisor driver. A value of 10 indicates a GS/OS "boot driver," a GS/OS driver which is loaded before other GS/OS drivers to control the boot device. For further information on boot drivers, contact Developer Technical Support. A value of 11 is reserved and must not be used by GS/OS device driver authors.

Bits 5 through 0 are defined by the GS/OS driver type. For standard device drivers, this field indicates the maximum number of devices supported; the GS/OS Device Dispatcher will use this field to allocate memory when the driver is loaded. For all other GS/OS driver types, this field is reserved and must not be used by GS/OS driver authors.

GS/OS driver definitions are documented in GS/OS Reference, Volume 2.

AppleTalk Drivers

The subclass is used by AppleTalk to determine in which order the drivers should be loaded. Programmers should treat every AppleTalk driver (all subclasses) as reserved; do not change the auxiliary type in any way, not even to deactivate the drivers.

Class $02 drivers are AppleTalk protocol drivers, including ROM patches for AppleTalk firmware. These drivers are currently loaded and initialized by the SCC.Manager supervisor driver.

Note: The SCC.Manager driver must not be deactivated as it arbitrates use of the serial ports. It is required for AppleTalk to function. Similarly, the AppleTalk drivers must not be deactivated individually or AppleTalk may not be initialized

correctly. Disable AppleTalk in the Control Panel. You may then use the Apple IIGS Installer to remove AppleShare if you wish to remove AppleTalk drivers and protocols from a disk.

AppleTalk drivers are loaded only if AppleTalk is enabled in the Control Panel. The drivers are initialized in alphabetical order; therefore, you should not change the names of existing AppleTalk drivers.

In the driver subclass, bit 7 indicates whether the flag is a standard AppleTalk protocol. For protocols that you write and ship, this bit must be set to 1. If you feel your driver should have this bit set as a standard protocol, contact Developer Technical Support. Bits 0-3 of the subclass indicate the maximum ROM version for which the driver should be loaded. For example, a driver with a value of 1 in this field will not be loaded on a machine with a ROM version greater than 1. A driver with a value of 3 will not be loaded on future Apple IIGS machines, but will be loaded on all current machines.

AppleTalk drivers are called to initialize themselves in full 16-bit native mode. The A register contains the current AppleTalk channel number (i.e., which port is being used for AppleTalk); the X register contains the ROM version; the Y register contains the AppleTalk firmware slot number. Drivers should return from initialization with the carry clear and zero in the accumulator if initialization was successful or carry set and an error code in A if initialization failed. You may assume that AppleTalk is active during AppleTalk driver initialization. Your driver should perform necessary installation tasks (such as opening as socket or adding routines to the dispatch table at $E1D600) during initialization.

The remaining bits in the driver subclass are reserved and must not be used by AppleTalk driver authors.

MIDI Tools Drivers

The subclass field for MIDI Tools drivers is currently reserved and should be set to zero. MIDI Tools Drivers are documented in Apple IIGS Technical Note #54, MIDI Drivers.

Third-Party Multimedia Drivers

The third-party multimedia driver class indicates drivers used by applications or other non-system components to control multimedia peripherals such as videodisc or video tape players. The subclass for this class is assigned for each driver by Developer Technical Support, as are most file type and auxiliary type combinations. Developers wishing to provide a multimedia driver must contact Developer Technical Support for a subclass assignment.

What's Reserved?

Since so many types of drivers are all using the same file type, it is essential that you adhere to the auxiliary type conventions specified in this Note. If you are creating a driver whose auxiliary type can not be completely defined using the guidelines in this Note, contact Developer Technical Support for assistance, or for further assignment. If you are creating a driver which does not fit into an existing driver class, contact Developer Technical Support for a new class assignment.

Do not use any field marked as reserved in this Note to store any number

other than zero.

Further Reference

- Apple IIGS Toolbox Reference
- GS/OS Reference, Volume 2
- Apple IIGS Technical Note #35, Printer Driver Specifications
- Apple IIGS Technical Note #36, Port Driver Specifications
- Apple IIGS Technical Note #54, MIDI Drivers

END OF FILE FTN.BB.xxxx

```
##########################################################################
### FILE: FTN.BC.xxxx
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $BC (188)
Auxiliary Types: All

Full Name: Apple IIgs Generic Load File
Short Name: Load file (generic)

Written by: Matt Deatherage July 1990

Files of this type and auxiliary type contain OMF for the Apple IIgs.

Files of type $BC contain data that is to be loaded by one of the Apple IIgs Loaders (either the System Loader or ExpressLoad). No other information about the data is known. Load files which do not fit other load file type definitions ($B3-$BD) should be of type $BC. The most common use for files of type $BC are code modules.

The following auxiliary type assignment is current for this file type as of the publication date of this Note:

Auxiliary Type	Short Name	Developer
$0000	Load file (generic)	any
$4001	Nifty List Module	DAL Systems

Table 1-Auxiliary Type Assignments

Note that auxiliary type $0000 is listed as "any" developer. You may create and use files of $BC and auxiliary type $0000 as you wish; Apple does not maintain a specific assignment for this auxiliary type. However, if you use auxiliary type $0000, you must not identify your files exclusively by it. Many load files can share this auxiliary type, and it alone is not suitable for file content identification. Apple allows the free use of auxiliary type $0000 since load files must have a load file type and often further identification is not necessary.

Also note that auxiliary type $0000 is reserved in other file types; you may not use auxiliary type $0000 in other file types without assignment from Developer Technical Support.

Further Reference

- GS/OS Reference

```
### END OF FILE FTN.BC.xxxx
```

```
##########################################################################
### FILE: FTN.BD.xxxx
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $BD (189)
Auxiliary Type: All

Full Name: GS/OS File System Translator
Short Name: GS/OS File System Translator

Written by: Matt Deatherage September 1990

Files of this type and auxiliary type contain file system translators for GS/OS.

Files of type $BD contains file system translators, or FSTs. FSTs do not load if bit 15 of their auxiliary type is set.

GS/OS calls FSTs to interpret the physical file systems stored on block devices. By asking translation software to read the file system, GS/OS can read virtually any file system while having only an abstract file system assumed in the operating system code. Not all released file system translators are required, saving space on disk and in memory.

The format for FSTs is Apple confidential and subject to change with every system software release; Apple will release all future FSTs for GS/OS. Third-party developers may not create GS/OS FSTs--no documentation is available, and disassembly of the code for this purpose is not permitted. This is not an easy decision for Apple, which is a company that was built upon and operates with the goal to empower individuals through computing. Not revealing information isn't exactly consistent with this goal. There are, however, reasons for this policy.

First, FSTs are not as modular as they could be. Some GS/OS level changes require changes to all of the FSTs to be implemented. These changes range in magnitude from internal system service call changes to adding new parameters to existing calls. GS/OS is not tolerant of FSTs that do not know about such changes. The FST structure is straightforward, but it is also complex enough that disassembly of existing FSTs does not cover all the bases.

Second, it can create chaos for users. Two file system translators for a file system is far worse than none at all. No physical file system exactly matches the GS/OS abstract file system, so every FST must have file system specific behavior. Although some of these behaviors are well documented (parameters that do not fit in the abstract file system go in the option_list, for example), no two independently-designed FSTs for the same file system can possibly do such things identically.

For example, if there were two third-party DOS 3.3 FSTs available, each would have its own FSTSpecific subcalls, option_list parameters and other implementation differences. Since there is only one file_sys_ID per file

system, programs that create correct data structures for one DOS 3.3 FST may blow up with the other one.

If users somehow manage to figure this out, the only way to change FSTs is to enter the *:System:FSTs folder, deactivate one FST, activate another one and reboot, which is not acceptable. Even switching FSTs is unacceptable for archival and copying programs which may have stored option_list parameters embedded in files. Futhermore, if the file system is bootable, that makes boot blocks and a file system stub which are also tied to an FST, and users would have a horrible time changing those.

The best solution to these problems for Apple's customers (who are also your customers) is for Apple to maintain control over the development of file system translators. Apple will provide file system translators for other file systems. If you have requests for how certain features of any file system should be handled by future FSTs, please contact Developer Technical Support.

Further Reference

- GS/OS Reference

END OF FILE FTN.BD.xxxx

```
##########################################################################
### FILE: FTN.C0.0000
##########################################################################

Apple II
File Type Notes
```

Developer Technical Support

File Type: $C0 (192)
Auxiliary Type: $0000

Full Name: Paintworks Packed Super Hi-Res Picture File
Short Name: Paintworks Packed Picture

Written by: Matt Deatherage November 1988

Files of this type and auxiliary type contain a packed Super Hi-Res graphics image created by Activision's Paintworks program.

Files of type $C0 and auxiliary type $0000 contain a packed Super Hi-Res graphics image created by Activision's Paintworks program. For more information on Paintworks, contact:

Mediagenic, Inc.
3885 Bohannon Drive
Menlo Park, CA 94025
Attn: Paintworks Technical Support

The format of the file is as follows:

+000 to +01F	Bytes	Super Hi-Res Palette
+020 to +021	Word	Background color
+022 to +221	Bytes	Patterns. 16 QuickDraw II patterns, each 32 bytes in length.
+ 222 to EOF	Bytes	Packed graphics data. Note that the unpacked data could be longer than one Super Hi-Res screen (Paintworks allows full-page sized documents).

```
### END OF FILE FTN.C0.0000
```

```
##########################################################################
### FILE: FTN.C0.0001
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $C0 (192)
Auxiliary Type: $0001

Full Name: Packed Apple IIGS Super Hi-Res Image File
Short Name: Packed Super Hi-Res Image

Written by: Matt Deatherage November 1988

Files of this type and auxiliary type contain a packed Apple IIGS Super Hi-Res screen image.

Files of type $C0 and auxiliary type $0001 contain a packed Apple IIGS Super Hi-Res screen image, which is created by passing the entire 32K Super Hi-Res screen area, including the scan line control bytes (SCB) and color tables, through the PackBytes routine.

If you restore a file of this type to its original 32K size with UnPackBytes, you can save the unpacked data to a file of type $C1 and auxiliary type $0000 (Apple IIGS Super Hi-Res Graphics Screen Image).

Further Reference
o Apple II File Type Notes, File Type $C1, Auxiliary Type $0000

END OF FILE FTN.C0.0001

```
##########################################################################
### FILE: FTN.C0.0002
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $C0 (192)
Auxiliary Type: $0002

Full Name: Apple IIgs Super Hi-Res Picture File
Short Name: "Apple Preferred" Picture

Revised by: Matt Deatherage December 1991
Written by: Steve Glass, Eagle Berns, Art Cabral & Pete McDonald October 1987

Files of this type and auxiliary type contain a Super Hi-Res picture in the Apple-recommended format.
Changes since September 1990: Added a recommeendation for more conventional respresentation of 3200-color pictures.

Files of type $C0 and auxiliary type $0002 contain a Super Hi-Res picture in the format recommended by Apple. The file consists of a series of variable-length blocks (in any order), each with the same general format:

Length	Long	Defines the length of this block, including Length itself.
Kind	String	String used to identify the type of block. The string is stored as a series of case-sensitive ASCII characters preceded by a length byte. To avoid confusion, the use of uppercase characters only is recommended.
Data	Bytes	Variable amount of block-specific data.

With this scheme, the file format is flexible and can be extended. Standard blocks which most applications want are defined below, however, these are not required. Individual applications can define other blocks. Application-dependent information can be stored to allow other applications to ignore it. Your application should simply ignore blocks it finds which it is not prepared to interpret.

Data Types

Integer	16-bit signed word.
LongInt	32-bit signed long word.
ColorEntry	16-bit word. The nibbles in the word are interpreted as RGB values as follows: $0RGB. The high nibble of the high byte must be zero and should be ignored when reading files. The low nibble of the high byte is the value for

	red, the high nibble of the low byte is the value for green, and the low nibble of the low byte is the value for blue (see Figure 16-19 on page 16-31 of the Apple IIgs Toolbox Reference).
ColorTable	16 words: array [0..15] of ColorEntry
ModeWord	16-bit word. The high byte determines the definition of the mode. If high byte = 0, then the low byte is the mode bit portion of the SCB for the scan line (see Figure 16-22 on page 16-34 of the Apple IIgs Toolbox Reference). Other bits are reserved and must be zero, as other modes are not yet defined.
DirEntry	A two-word structure used to define the characteristics of each packed line: Integer: Number of bytes to unpack ModeWord: Mode
PatternData	32 bytes of pattern information

MAIN Information Block

Every file usually, but not necessarily (i.e., a file of palettes only), includes a MAIN block.

Length	LongInt
Kind	String "MAIN"
MasterMode	ModeWord (from the MasterSCB of QuickDraw II. When reading a file, this word should be used in a SetMasterSCB call.)
PixelsPerScanLine	Integer (must not be zero)
NumColorTables	Integer (may be zero)
ColorTableArray	[0..NumColorTables-1] of ColorTable
NumScanLines	Integer (must not be zero)
ScanLineDirectory	[0..NumScanLines-1] of DirEntry
PackedScanlines	[0..NumScanLines-1] of Packed Data (Obtained by performing a PackBytes call on the pixel image of a single scan line.)

PATS Information Block

The PATS block contains patterns which may be associated with the picture.

Length	LongInt
Kind	String "PATS"
NumPats	Integer
PatternArray	[0..NumPats-1] of PatternData

SCIB Information Block

The SCIB block contains information relating to the current drawing pattern for the document. This information is used by paint programs that want to save a foreground pattern, a background pattern, and a frame pattern with the image.

Length	LongInt
Kind	String "SCIB"

ForegroundPattern	PatternData
BackgroundPattern	PatternData
FramePattern	PatternData

PALETTES Information Block

The PALETTES block contains information on the color tables. Its use is intended for color table files. If the file being saved contains a pixel image, then the color tables associated with that picture should be saved in the MAIN block, and this block would not be used.

Length	LongInt
Kind	String "PALETTES"
NumColorTables	Integer (must not be zero)
ColorTableArray	[0..NumPalettes-1] of ColorTable

Other Information Blocks

Apple Preferred Format is an extensible graphics file format. Since its release, some developers have contributed other block definitions that other developers may find to be useful. Please feel free to incorporate these blocks into Apple Preferred files, but you must be prepared to deal with Apple Preferred files that do not contain these additional blocks.

MASK Information Block

The MASK block contains information on which portions of a graphic image should be modified. The structure is similar to that of the MAIN block. However, the MASK array of PackedScanLines contains zeroes where no drawing is to occur (where the image is transparent) and ones where drawing may occur (where the image is solid). The structural similarity to the MAIN block can help by allowing some of your code to do double work.

Length	LongInt
Kind	String "MASK"
MasterMode	ModeWord (from the MasterSCB of QuickDraw II. When reading a file, this word should be used in a SetMasterSCB call.)
PixelsPerScanLine	Integer (must not be zero)
NumColorTables	Integer (must be zero)
NumScanLines	Integer (must not be zero)
ScanLineDirectory	[0..NumScanLines-1] of DirEntry
PackedScanlines	[0..NumScanLines-1] of Packed Data (Obtained by performing a PackBytes call on the pixel image of a single scan line.)

Note: There is no ColorTableArray, as indicated by a zero value in NumColorTables.

Note: The scan lines to be packed should only contain mask values of one and zero.

MULTIPAL Information Block

The MUTLIPAL block contains extra color tables necessary for displaying pictures that contain up to 3,200 colors on the screen.

Length	LongInt
Kind	String "MULTIPAL"
NumColorTables	Integer (should be the same as NumScanLines in MAIN). This is typically 200, but any value is legal.
ColorTableArray	[0..NumColorTables-1] of ColorTable. These are in the regular (0-15) order.

If you use the MULTIPAL block to store pictures with more colors than are typically displayable on the screen, Apple recommends you also create a MAIN block with a 16-color (or grayscale) representation of the picture, so users may open these files in les specialized applications to at least preview the picture enclosed.

Further Reference

- Apple IIgs Toolbox Reference
- Apple IIgs Technical Note #94, Packing It In (and Out)

END OF FILE FTN.C0.0002

```
##########################################################################
### FILE: FTN.C0.0003
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $C0 (192)
Auxiliary Type: $0003

Full Name: Packed Apple IIGS QuickDraw II Picture File
Short Name: Packed QuickDraw II PICT File

Written by: Matt Deatherage November 1988

Files of this type and auxiliary type contain a packed QuickDraw II picture.

Files of type $C0 and auxiliary type $0003 contain a packed QuickDraw II picture (a.k.a. PICT, after its counterpart on the Macintosh), and this file format is the same as file type $C1 and auxiliary type $0001, except files in this format are packed. If you encounter a file of this type, you should be able to get its length, allocate a handle of the same size, read the file into the handle, use UnPackBytes to restore the picture to its original format, and call DrawPicture to display the picture. Refer to Apple IIGS Technical Note #46, DrawPicture Data Format for more information on the internal format of QuickDraw II pictures.

Further Reference
- Apple IIGS Toolbox Reference Manual, Volume 2
- Apple IIGS Technical Note #46, DrawPicture Data Format

END OF FILE FTN.C0.0003

```
##########################################################################
### FILE: FTN.C1.0000
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $C1 (193)
Auxiliary Type: $0000

Full Name: Apple IIGS Super Hi-Res Graphics Screen Image
Short Name: Super Hi-Res Screen Image

Written by: Matt Deatherage November 1988

Files of this type and auxiliary type contain a 32K unpacked picture image.

Files of type $C1 and auxiliary type $0000 contain a 32K unpacked Super Hi-Res screen image, which is created by writing the entire Super Hi-Res screen area ($E12000-$E19FFF) to a file. If you pass this data through the PackBytes routine, you can save the result as a file of type $C0 and auxiliary type $0001 (Packed Apple IIGS Super Hi-Res Image File).

Note: The first release of Activision's PaintWorks assumes that palette colors are ordered from highest to lowest luminance.

Further Reference
o Apple II File Type Notes, File Type $C0, Auxiliary Type $0001

END OF FILE FTN.C1.0000

```
##########################################################################
### FILE: FTN.C1.0001
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $C1 (193)
Auxiliary Type: $0001

Full Name: Apple IIGS QuickDraw II Picture File
Short Name: QuickDraw PICT File

Written by: Matt Deatherage November 1988

Files of this type and auxiliary type contain an unpacked QuickDraw II picture.

Files of type $C1 and auxiliary type $0001 contain an unpacked QuickDraw II picture (a.k.a. PICT, after its counterpart on the Macintosh), and this file format is the same as file type $C0 and auxiliary type $0003, except files in this format are unpacked. If you encounter a file of this type, you should be able to get its length, allocate a handle of the same size, read the file into the handle, and call DrawPicture to display the picture. Refer to Apple IIGS Technical Note #46, DrawPicture Data Format for more information on the internal format of QuickDraw II pictures.

Further Reference
- Apple IIGS Toolbox Reference Manual, Volume 2
- Apple IIGS Technical Note #46, DrawPicture Data Format

END OF FILE FTN.C1.0001

```
##########################################################################
### FILE: FTN.C1.0002
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $C1 (193)
Auxiliary Type: $0002

Full Name: Super Hi-Res 3200 color screen image
Short Name: Super Hi-Res 3200 color image

Written by: Matt Deatherage September 1990

Files of this type and auxiliary type contain pictures with up to 3,200 colors displayed simultaneously.

Files of type $C1 and auxiliary type $0002 contain Super Hi-Res screen images with extra color tables so that up to 3,200 colors can be displayed simultaneously.

Such a display technique is visually striking but is not easy--the program must watch the Mega II video counters and change the color tables in the palette for future scan lines while the Apple IIgs is drawing a different scan line. All such pictures use 320 mode.

This format is for uncompressed 3,200-color pictures. It is often referred to as "Brooks format" after the designer of the format, John Brooks. For a way to store these pictures compressed, see the MULTIPAL block in the Apple Preferred format.

Definition

The following definition is used in this document in addition to those defined for all Apple II file types:

Color table — A table of sixteen two-byte entries, where each entry in the table is a master color value ($0RGB, where R is the red component, G is the green component and B is the blue component).

File Structure

The format for these files is similar to that for Super Hi-Res screen images.

pixelData	(+000) 32000 Bytes	Pixel data to be displayed on the Super Hi-Res screen.
colorTables	(+32000) 200 Color Tables	One color table for each scan line. Each color table is stored in reverse order; the color value for color 15 is stored first.

Further Reference

- Apple IIgs Toolbox Reference, Volume 2
- Apple IIgs Hardware Reference
- File Type Note for file type $C0, auxiliary type $0002, Apple Preferred Format

END OF FILE FTN.C1.0002

```
##########################################################################
### FILE: FTN.C7.XXXX
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $C7 (199)
Auxiliary Type: All

Full Name: Control Panel
Short Name: Control Panel

Revised by: Dave "Flag Bits" Lyons May 1992
Written by: Matt Deatherage & Darryl Lovato September 1989

Files of this type contain control panels (formerly called CDevs) for the Apple IIgs Control Panels New Desk Accessory.

CHANGES SINCE DECEMBER 1991: Updated for Control Panels NDA 2.0 in System 6.0.

Files of type $C7 contain control panels. When deciding whether to write an NDA or a control panel, keep in mind that control panels normally don't take any RAM when they are not in use, don't take up space in the Apple menu, and automatically re-open at the same screen position where they were last used. On the other hand, control panels are limited to a single code segment, and the main window has a fixed size.

Before System 6, the Control Panels NDA presented control panels (then called "CDevs" to the user) in a single window. In System 6, each control panel appears in its own window.

AUXILIARY TYPE

The auxiliary type of CDevs is defined bit by bit. Currently, only bit 15 is defined--it's the "inactive" bit. As with desk accessories, FSTs, and setup files, the control panel is not loaded or used if this bit is set. All other bits are reserved and must be set to zero. (In 6.0, inactive control panels do not appear in the Control Panels NDA's list, but the user can still open them directly from the Finder.)

HOW CONTROL PANELS WORK

The Control Panels NDA lets the user choose control panels, and it communicates with open control panels using a small collection of messages. Most events the Control Panels NDA receives from the system are handled by calling TaskMasterDA. At certain times, the Control Panels NDA sends messages to control panels.

The Control Panels NDA takes care of nearly everything necessary, including tracking controls. Control panel windows are usually just windows full of

extended controls; the control panel receives a HitCDEV message every time the user adjusts the value of one of the controls.

EVERY control in the control panel window must be an extended control. Older, non-extended controls are not allowed; all controls MUST be created with NewControl2. When one of these controls is "hit," the Control Panels NDA calls the control panel with the HitCDEV message, the control handle, and the control ID. This allows the control panel to respond to user actions. User interface items beyond extended controls (for example, modal windows) must be handled entirely by the control panel (that is, the Control Panels NDA is not involved).

NOTE: Setting the fInWindowOnly bit of Pop-up menu controls is not recommended.

THE CONTROL PANELS WINDOW

In version 1.0 of the Control Panel NDA there is a single window, and exactly one control panel is always active in a portion of that window.

With System 6, this is no longer true. Many control panel messages include "the control panel's window pointer" as one of the parameters. This is guaranteed to be the window containing the control panel's controls, but little else is guaranteed.

For example, do not draw outside the area containing your control panel's controls; do not compute other window sizes from the size of this window; and do not assume that the Control Panels NDA will offset your controls' coordinates by the same amount version 1.0 did.

Do not hard-code any window coordinates. The Control Panels NDA shifts all your controls by some amount horizontally and vertically, but this amount will not stay the same between different versions of the Control Panels NDA (it can be zero). If you draw things besides controls in the window, compute the coordinates relative to a control on the fly.

In System 6.0, each control panel gets its own window.

RESOURCE FORK

The Control Panels NDA opens your control panel's resource fork differently depending on whether the machine was booted from an AppleShare file server or from a local volume.

When the machine was booted from AppleShare, your resource fork is opened with read-only access so that more than one user can have your control panel open at once. When the machine was booted locally, your resource fork is opened with "as allowed" access (this means you will have read/write access if the control panel file is unlocked and was not already opened read-only by some other part of the system).

When your control panel receives the BootCDEV message at boot time, its resource fork is always open read-only.

FILE FORMAT

A control panel is defined by three resources (additional resources may be

present). The data fork is normally empty, but a control panel that requires System 6 or later may put code in the data fork in OMF format (it's up to the control panel to determine its own pathname and use InitialLoad2 to load code from the data fork--a control panel can find its pathname by using GetCurResourceFile, GetOpenFileRefNum, and GetRefInfo).

The three required resources are the CDev code resource (type rCDEVCode=$8018, ID=$00000001), the CDev flags resource (type rCDEVFlags=$8019, ID $00000001) and the CDev's icon (type rIcon=$8001, ID=$00000001).

It is a good idea to make sure each released version of your control panel file has a different creation date, since the system caches certain information about your control panel in the CDev.Data file. The system uses the creation date to notice that a new version of your CDev is present.

You may also want to delete the *:System:CDevs:CDev.Data file, if it exists, as part of your CDev installation process.

THE ICON RESOURCE

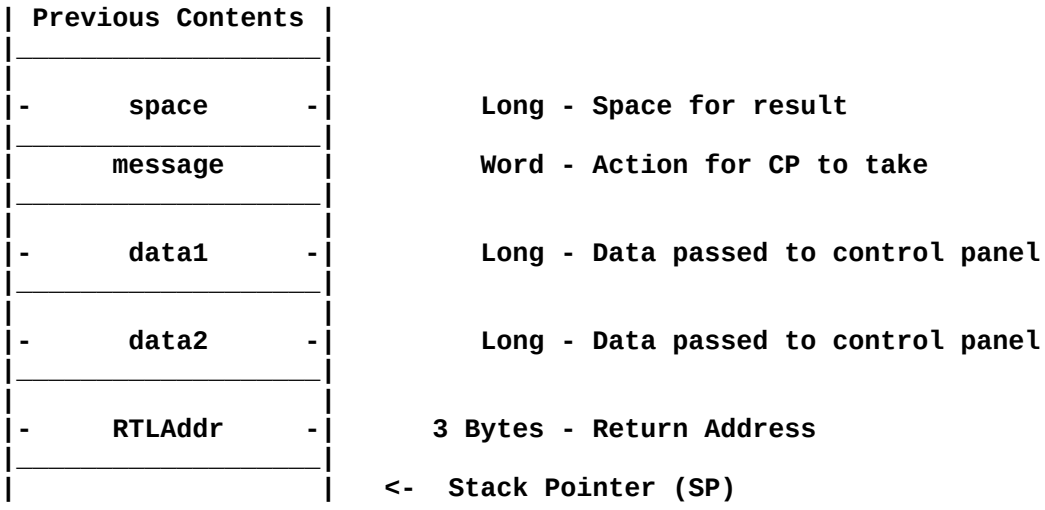
Each control panel's icon is a standard icon resource. This icon appears in the Control Panels window; it is also displayed at boot time if the CDev has any initialization code (described later).

If the icon is to be displayed during boot time (before System 6.0), it must be exactly 28 pixels wide. In 6.0, this restriction is gone, but 28 is still a nice width.

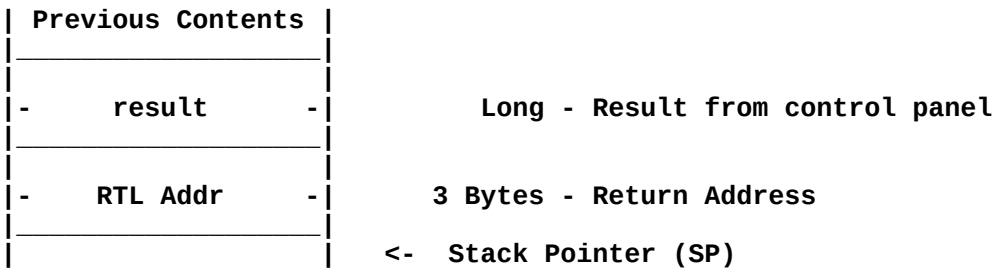
THE CDEV CODE RESOURCE

The rCDevCode(1) resource contains code to do the real work. A code resource has the same format as an OMF load file; the code resource converter (which is part of the system) is responsible for loading code resources. Eventually, InitialLoad2 loads the code from memory. This process gives the rCDevCode resource a maximum size of 64K.

When the control panel code gets control, the stack is as follows:



The control panel must remove the input parameters from the stack and perform an RTL, so the calling routine may then pull the four-byte result parameter off the stack. Just before the control panel code RTLs, the stack must be formatted as follows:



This function, like nearly all toolbox functions, is a "Pascal" function, and may be declared in Pascal as follows:

```
function MyControlPanel(message: Integer; data1, data2: Longint): LongInt;
```

It may be declared in C as follows:

```
pascal Long MyControlPanel(message, data1, data2)
int message;
long data1, data2;
```

Data1 and Data2 depend on the value of message; message is the parameter that tells the CDev code what needs to be done. Higher-level language control panels can easily be arranged as a giant switch (or case, as the case may be) statement.

There are twelve defined "CDev" messages. Where parameters are not listed, they are undefined.

MESSAGE 1: MachineCDEV

The Control Panels NDA always compares the Apple IIgs ROM version against the minimum ROM version you put in the CDev Flags resource. If the machine's ROM version is too low, the control panel does not appear (and cannot be opened).

The MachineCDEV message was not supported before System 6.0. In 6.0, if the wantMachine bit is set in the CDev Flags resource, the control panel receives MachineCDEV when the user attempts to open it. The input parameters are undefined. Return a nonzero result to allow the open, or return zero to abort the open. When returning zero, you may want to display an alert explaining why the control panel cannot be opened.

MESSAGE 2: BootCDEV

If the wantBoot flag is set in the CDev Flags resource, this routine is called during the IIgs boot sequence. The parameters are undefined before 6.0. In 6.0, data1 is defined to point to a data word that is initialially zero. If you set bit 0 of this word while handling the BootCDEV message, the Control Panels NDA will draw an "X" over your icon (but it will not call SysBeep2 for

you; do that yourself if appropriate).

BootCDEV is called only during a real boot--it doesn't get control on a switch back to GS/OS from ProDOS 8. The Control Panels NDA draws the icon (from the icon resource) on the boot screen. (Before 6.0, the icon must be exactly 28 pixels wide if it is drawn at boot time.)

At best, the machine state during this call can be termed bad. QuickDraw II is not even available. Be sure to save and restore any system resources you use, including the data bank register and the direct page register.

NOTE: If your CDev expects to receive a BootCDEV message, it should still behave gracefully if BootCDEV was never received and the user attempts to use the control panel (for example, tell the user to put the file into the CDevs folder and restart the system).

In System 5.0.x, the user could drag your control panel into the CDevs folder and then try to use it without restarting. In System 6.0, control panels are directly launchable from the Finder, but only the ones in the CDevs folder receive BootCDEV messages.

MESSAGE 3: Reserved

This message is reserved for future use as a shutdown message.

MESSAGE 4: InitCDEV

If the wantInit flag is set in the CDev Flags resource, this routine is called with data1 equal to the control panel's window pointer. When InitCDEV is called, CreateCDEV (message 7) has already been called. Controls should have been created in CreateCDEV, and this routine is an ideal place to initialize the controls before they are displayed.

MESSAGE 5: CloseCDEV

This routine is called if the wantClose bit is set in the CDev Flags resource. If so, CloseCDEV is called when your control panel is closing. This is a good place to dispose of any memory you allocated or to save settings that need to be saved. The disposal of the control panel's controls is handled by the Control Panels NDA. The window pointer is in data1.

MESSAGE 6: EventsCDEV

If the wantEvents bit is set in the CDev Flags resource, the Control Panels NDA calls this routine with data1 as a pointer to the event record (this is an Event Manager event record, not a TaskMaster-style task record). The window pointer is in data2. The Control Panels NDA, like all NDAs, is passed events, which the it then handles by using the TaskMasterDA call. This routine is called before TaskMasterDA is called, so the control panel can change the event record before the Control Panels NDA handles it.

MESSAGE 7: CreateCDEV

This routine is only called if the wantCreate bit is set in the CDev Flags resource. When called, the control panel's window pointer is in data1. The control panel must create any controls it has during this call. The control

panel's resource fork is open during this call, so Resource Manager calls may be made (and controls may be created from resources in the control panel file). All control rectangles are relative to the upper-left corner of the part of the window a control panel owns (in 6.0 this happens to be the whole window). The Control Panels NDA handles setting the offsets of the controls to the proper place in the window. Initialization of the controls must be done in the InitCDEV call.

If the wantCreate bit is not set, the control panel must contain an rControlList (type=$8003) resource with ID $00000001. The Control Panels NDA automatically creates your controls from the resource.

MESSAGE 8: AboutCDEV

If the wantAbout bit is set in the CDev Flags resource, the Control Panels NDA calls this routine when the user selects "Help" while your control panel's icon is selected. The window pointer to the help window is in data1. The Control Panels NDA takes care of the icon, author, version string and the "OK" button. The easiest way to handle help is simply to create a static text control with the help text in it.

If the wantAbout bit is not set, your control panel must have an rControlList resource with ID $00000002. When the user selects "Help" while your control panel's is selected, the Control Panels NDA uses this resource to create your additional About controls.

NOTE: In 6.0, when a control panel receives the AboutCDEV message, the Font Manager and TextEdit are always started. The Control Panels NDA can display a control panel's About box without ever opening the control panel. Making TextEdit available avoids a potential incompatibility with some control panels (such as General) that start up TextEdit on receiving AboutCDEV, assuming they will have a chance to shut it back down later, on receiving CloseCDEV.

MESSAGE 9: RectCDEV

Normally, the Control Panels NDA uses the rectangle in the CDev Flags resource for the control panel's display rectangle. However, if the wantRect bit is set in the CDev Flags resource, this routine is called before the control panel is displayed with data1 containing a pointer to the display rectangle. The rectangle may be modified by this routine. This gives control panels the chance to use different sized rectangles for different occasions. For example, on ROM 03, the serial port control panels show fewer parameters when the port is set to AppleTalk (since fewer parameters are changeable). In that instance, the RectCDEV routine changes the rectangle to be smaller.

MESSAGE 10: HitCDEV

If the CDev wants to know when a control has been hit, it can set the wantHit bit in the CDev Flags resource. When called, the handle to the control in question is in data1 and that control's ID is in data2. The control panel may then take action based upon the control selection.

If you need the window pointer, you can get it from the ctlOwner field of the control record handle in data1.

Note: If your control panel contains any extended List controls, the toolbox automatically creates a scroll bar control for each list. These scroll bars are standard (not extended) controls; this is the exception to the rule that all control panel controls must be extended. When the user tracks the scroll bar, the HitCDEV data1 parameter is a valid control handle, but data2 is an unpredictable large value (because no control ID is available for a non-extended control). In 6.0, the control ID returned in this case is always $FFFFFFFF.

MESSAGE 11: RunCDEV

This routine is called if the wantRun bit in the CDev flags resource is set. It enables control panels to receive a call as often as the Control Panels NDA receives run events from SystemTask (currently once per second).

The control panel's window pointer is in data1. (This is true even before 6.0, but it was not previously documented.)

MESSAGE 12: EditCDEV (6.0 and later)

This routine is called if the wantEdit bit in the CDev flags resource is set, when the user chooses Undo, Cut, Copy, Paste, or Clear from the Edit menu (if the items have the proper item numbers), and when the user types Command-Z, -X, -C, or -V.

The control panel's window pointer is in data2. The low word of data1 indicates what kind of edit operation is happening. The codes are the same as what SystemEdit passes to NDAs (Toolbox Reference 1, page 5-7):

$0005	Undo
$0006	Cut
$0007	Copy
$0008	Paste
$0009	Clear

All other codes are reserved for future use.

THE CDEV FLAGS RESOURCE

The CDEV Flags resource tells the Control Panels NDA which messages the control panel accepts. It also tells the Control Panels NDA certain things about the operating environment required for the CDev.

flags (+000) Word The flags word tells the Control Panels NDA which messages (defined in the discussion of the rCDevCode resource) the control panel wants:

Bits 15 - 12: Reserved, must be zero.
Bit 11: wantEdit
Control panel wants edit events.
Bit 10: wantRun
Control panel wants run events.
Bit 9: wantHit
Control panel wants control hits.
Bit 8: wantRect
Control panel wants rectangle messages.

		Bit 7:	wantAbout Control panel wants "about" messages.
		Bit 6:	wantCreate Control panel wants create messages.
		Bit 5:	wantEvents Control panel wants event records.
		Bit 4:	wantClose Control panel wants close messages.
		Bit 3:	wantInit Control panel wants initialization message.
		Bit 2:	wantShutDown Reserved, must be zero.
		Bit 1:	wantBoot Control panel wants boot messages.
		Bit 0:	wantMachine Control panel wants machine messages (6.0).
enabled	(+002)	Byte	If this value is zero, the control panel is never activated. NOT USED.
version	(+003)	Byte	An integer version number assigned by the author.
machine	(+004)	Byte	This byte contains a minimum ROM version required for the control panel. For most control panels this is 1, but some (requiring, for example, hardware text page two shadowing) want 3 in this byte.
reserved	(+005)	Byte	Reserved, must be zero.
data rect	(+006)	4 Words	QuickDraw II rectangle within which the control panel is displayed. The top left of this rectangle must be (0,0).
name	(+014)	16 Bytes	A string (Pascal) giving the name of the control panel. Names longer than 15 bytes are not allowed. Note that this field requires 16 bytes regardless of the string length.
author	(+030)	33 Bytes	A string (Pascal) giving the name of the control panel's author. Names longer than 32 bytes are not allowed. Note that this field requires 33 bytes regardless of the string length.
version	(+063)	9 Bytes	A string (Pascal) giving the version of the control panel. Strings are typically of the format "v1.0". Version strings longer than eight bytes are not allowed. Note that this field requires nine bytes regardless of the string length.

OPENING ADDITIONAL RESOURCE FILES

The Control Panels NDA, not any individual control panel, owns the Resource Manager search path that is in effect when a control panel routine gets control. While handling a message, you may temporarily open additional resources files in the same search path, but you must close them and call SetCurResourceFile to its previous value before returning control to the Control Panels NDA.

There may be extra resource files in the search path that you know nothing about, so do not assume that your extra file is adjacent to your control panel's resource file in the search path.

COLOR TABLE SWAPPING

Since control panels generally assume the sixteen standard 640-mode dithered colors are available, Control Panels NDA 2.0 automatically provides a standard color table whenever the "Control Panels" window or any individual control panel window is in front. (It ought to do the same thing for the Help and credits windows, but it does not.)

The color table provided in 640 mode is identical to the default 640-mode color table.

The color table provided in 320 mode provides colors almost identical to the default 640 colors. This is not the same as the default 320-mode color table. (See Apple IIgs Technical Note #63, Table 3.)

PROGRAMMATIC INTERFACE TO THE CONTROL PANELS NDA

You can use SendRequest in the System 6 Tool Locator to ask the Control Panels NDA to do two things for you: Open the main window, or open a control panel from a pathname.

You must send the requests by name to "Apple~Control Panel~".

Request code $9001 is cpOpenControlPanels. dataIn is reserved and must be zero.

Request code $9000 is cpOpenCDev. dataIn and dataOut are as defined for the finderSaysOpenFailed request (see the Finder 6.0 documentation). You can also open a control panel by pathname by sending finderSaysBeforeOpen, as permitted in the Finder documentation.

Further Reference

- Apple IIgs Toolbox Reference, Volumes 1-3
- System 6.0 Documentation

END OF FILE FTN.C7.XXXX

```
##########################################################################
### FILE: FTN.C8.0000
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $C8 (200)
Auxiliary Type: $0000

Full Name: Apple IIGS Font File
Short Name: Font

Written by: Matt Deatherage **January 1989**

Files of this type and auxiliary type contain QuickDraw II fonts.

The Apple IIGS Font Manager looks in the SYSTEM/FONTS subdirectory of the boot volume for files with the file type $C8 and auxiliary type $0000. If it finds files of this type and auxiliary type, the Font Manager opens them as fonts. Files of this type and auxiliary type which are not fonts, but are in the SYSTEM/FONTS subdirectory, are opened as fonts, and can cause a system crash. However, the Font Manager does not treat files in this subdirectory as fonts if their auxiliary type is not $0000.

The format of a font file is as follows:

String	**Font name. A Pascal string containing the font family name of the font, as seen in the ChooseFont dialog.**
Font	**Immediately following the string is an Apple IIGS QuickDraw II Font definition, as documented beginning on page 16-41 of the Apple IIGS Toolbox Reference, Volume 2.**

The only difference between a font in memory and a font file on disk is the font family name, as a Pascal string, preceding the font definition on disk.

Further Reference

- **Apple IIGS Toolbox Reference, Volume 2**

END OF FILE FTN.C8.0000

```
##########################################################################
### FILE: FTN.CA.xxxx
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $CA (202)
Auxiliary Type: Reserved

Full Name: Finder Icons File
Short Name: Icons

Written by: Matt Deatherage July 1989

Files of this type and auxiliary type contain icons.

The Apple IIGS Finder keeps the icons it uses in files of type $CA. The Finder searches for these files in a directory named Icons on each volume mounted. Each icon contains information not only describing the icon and its mask (both regular and "small icon" sizes), but also information to match the icon to files from their file type, auxiliary type, and filename.

The Finder first attempts to load the file Finder.Icons from the Icons directory on the boot disk, stopping with a fatal error if it is not present (this file contains icons for devices as well as the icon to match files with no other icon). It then loads other icon files from that directory, and then from other disks.

The format of icon files is as follows:

+000	iBlkNext	Long	When loaded by the Finder, this is the handle to the next icon file (a linked list terminated by zero). On disk, this field should be zero.
+004	iBlkID	Word	ID number of this type of icon file. This field must be $0001 for the Finder to recognize the icon file.
+006	iBlkPath	Long	When loaded by the Finder, this is the handle to the pathname of this icon file. On disk, this field should be zero.
+010	iBlkName	16 Bytes	A 16-byte String of the name of the icon file.
+026	iBlkIcons	IconData	A list of Icon Data records.

The format of Icon Data records is as follows:

+000	iDataLen	Word	The length of this Icon Data record. A value of zero in this field terminates the list of Icon Data records.
+002	iDataBoss	64 Bytes	The pathname of the application that owns this icon. If this String has non-zero length, and the file

			this icon is associated with is a document (not an application, folder, device, or trash can), the Finder attempts to launch a file with this pathname when this icon is opened or printed. This string should be empty for non-documents. This is a full pathname, and most developers creating icons will wish to set this to the full pathname of the application on the shipping disk.
+066	iDataName	16 Bytes	A 16-byte String containing a file name. Files on disk must match the specification of this string or this icon will not be displayed for the files. The asterisk (*) serves as a wildcard character. For example, the string *.ASM matches all filenames ending with the characters .ASM.
+082	iDatatype	Word	File type associated with this icon. Files on disk must have this file type for this icon to be displayed. A file type of $0000 in this field matches any file type on disk. As an example, an application icon would want to have the filename of the application in the iDataName field and the file type $00B3 (GS/OS Application) in this field. Without the file type specification, the icon would show up for any file with the application's file name, including a folder on a hard disk in which the user has placed the application.
+084	iDataAux	Word	Auxiliary type associated with this icon. Similar to the file type field, a value of $0000 here matches any auxiliary type on disk.
+086	iDataBig	Icon	The normal size icon image data.
	iDataSmall	Icon	The small size icon image data.

The format of Icon records is the same as that listed in the QuickDraw II Auxiliary chapter of the Apple IIGS Toolbox Reference Manual. Previous icon structure documentation stated the iconType field of the Icon record (also known as the imType field) had to be zero. This is no longer true; the Finder respects color icons (bit 15 of iconType set) by not coloring the icon in funny ways, even if the user asks for it. The Finder still does this if bit 15 of iconType says the icon is a black-and-white icon.

Further Reference

- Apple IIGS Toolbox Reference, Volume 2
- Apple IIGS Icon Editor

END OF FILE FTN.CA.xxxx

```
##########################################################################
### FILE: FTN.D5.0007
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $D5 (213)
Auxiliary Type: $0007

Full Name: SoundSmith Music Sequence
Short Name: SoundSmith document

Written by: Matt Deatherage March 1990

Files of this type and auxiliary type contain music sequences used by SoundSmith.

SoundSmith is a music sequencing program that uses the full sound capabilities of the Apple IIGS. SoundSmith uses standard Apple Sampled Instrument format (ASIF) instruments to produce rich stereo sound with a variety of instruments.

For more information on SoundSmith, contact:

Huibert Aalbers
Travesía Andrés Mellado, 3
28015 Madrid
Spain
Attn: SoundSmith Technical Support
Phone: (34)-1-5446940

The File Format

SoundSmith sequences consist of a 600 byte header, followed by three equally-sized blocks containing the notes in the sequence, the effects to be applied to the notes, and parameters for the effects. The file concludes with 30 bytes of stereo information.

The Header

signature	(+000)	6 Bytes	ASCII bytes "SONGOK". An identifier to SoundSmith that the file is not corrupted.
length	(+006)	Word	The length of each of the three equally-sized blocks that follow the header (Main, Effects1, and Effects2).
tempo	(+008)	Word	The tempo for the song. A note is played each tempo/50th of a second (see the "Playing the Music" section in this Note).
instBlock1	(+020)	InstBlock	The instrument parameters for the first instrument.
instBlock2	(+050)	InstBlock	The instrument parameters for the second

			instrument.
instBlock3	(+080)	InstBlock	The instrument parameters for the third instrument.
instBlock4	(+110)	InstBlock	The instrument parameters for the fourth instrument.
instBlock5	(+140)	InstBlock	The instrument parameters for the fifth instrument.
instBlock6	(+170)	InstBlock	The instrument parameters for the sixth instrument.
instBlock7	(+200)	InstBlock	The instrument parameters for the seventh instrument.
instBlock8	(+230)	InstBlock	The instrument parameters for the eighth instrument.
instBlock9	(+260)	InstBlock	The instrument parameters for the ninth instrument.
instBlock10	(+290)	InstBlock	The instrument parameters for the tenth instrument.
instBlock11	(+320)	InstBlock	The instrument parameters for the eleventh instrument.
instBlock12	(+350)	InstBlock	The instrument parameters for the twelfth instrument.
instBlock13	(+380)	InstBlock	The instrument parameters for the thirteenth instrument.
instBlock14	(+410)	InstBlock	The instrument parameters for the fourteenth instrument.
instBlock15	(+440)	InstBlock	The instrument parameters for the fifteenth instrument.
musLength	(+470)	Word	Length of the music in SSBlocks.
musList	(+472)	128 Bytes	List of SSBlocks to play. Each block is identified by one byte (i.e., 0 3 5 2 2 n means play block 0, block 3 block 5, block 2, block 2, and block n respectively).

An SSBlock is 896 Bytes (64 * 14 bytes). The Main block is composed if SSBlocks. An InstBlock is a 30-byte block of instrument parameters defined as follows:

instName	(+000)	String	ASCII name of the instrument to be used. If this is less than 22 bytes (21 characters plus the length byte), it must be padded to take 22 bytes.
reserved	(+022)	Word	Reserved, set to zero.
volume	(+024)	Word	Volume for this instrument. Although this is a word parameter, legal values range from 0 to 255.
reserved	(+026)	Word	Reserved, set to zero.
reserved	(+028)	Word	Reserved, set to zero.

The Main block

The main part of the file consists of three equally-sized blocks. The length of each of the three parts is given by the Length field in the header; the entire Main block is 3*Length bytes long. Bytes in each block are related to each other positionally. For example, the first byte of the Effects1 and Effects2 blocks contain the effects to be applied to the note in the first byte of the Notes block.

The first block is the Notes block. Each byte is a MIDI Note number representing the note to play.

The second block is the Effects1 block. The high nibble of each byte determines which instrument should be used to play the note in the corresponding byte of the Notes block. The low nibble of each byte contains a value to be used by each effect.

The third block is the Effects2 block, and contains values to be used for the effects listed in the bytes of the Effects1 block.

Table 1 contains currently defined values for the effects and their values. All values not listed are reserved and must not be used.

Effects1 byte	Effects2 byte
0 = Arpegiatto	0 = no arpegiatto, $xy = increment1 of x, increment2 of y
3 = Set Volume	new volume ($00 - $FF)
5 = Decrease Volume	volume to subtract from instrument volume
6 = Increase Volume	volume to add to instrument volume
F = Set Tempo	new tempo

Table 1-SoundSmith Effects

Stereo Data

The file ends with 30 bytes of stereo data. The data is in 15 words, one for each instrument. A value of $0000 indicates the instrument uses the right channel; a value of $FFFF indicates the left channel. The first word corresponds to the first instrument, and so on.

Playing the Music

Those wishing to play the music in a SoundSmith file should use an interrupt-driven playback routine. The routine should be called every tempo/50th of a second. When called, the routine should read the next fourteen notes, Effects1 and Effects2 bytes, and play them on voices 1 through 14 using the specified instruments. Since SoundSmith provides 14 voices, you can use the fifteenth DOC oscillator as a timer to generate the required 50 Hz interrupts. When the note value is zero, you should do nothing (do not stop the sample). When the note value is 128 ($80), stop the sample on that voice.

Further Reference

- Apple IIGS Toolbox Reference, Volume 3

END OF FILE FTN.D5.0007

```
##########################################################################
### FILE: FTN.D5.xxxx
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $D5 (213)
Auxiliary Type: All

Full Name: Music Sequence File
Short Name: Music Sequence

Written by: Matt Deatherage January 1990

Files of this type and auxiliary type contain musical sequences.

Files of type $D5 contain data that is to be interpreted as a sequence of musical notes. A musical sequence can take several forms. It can be the data necessary for a music program to recreate the sequence aurally or visually; it can be information that is fed through sequencing hardware to produce the appropriate sounds; it can be a list of resource numbers that give a program the necessary means to recreate a sequence of music. The possibilities are virtually limitless. The most common use of sequences is to reproduce music aurally (through sound hardware internal or external to the system) or visually (to produce music notation on a screen or on paper).

The following auxiliary type assignments are current for this file type as of the publication date of this Note:

Auxiliary Type	Short Name	Developer
$0000	Music Construction Set song*	Electronic Arts
$8002	Diversi-Tune Sequence	DSR
$8003	Master Tracks Jr. sequence	Passport
$8004	Music Writer document	PyGraphics

Table 1-Auxiliary Type Assignments

The auxiliary types for this file type are reserved; any not listed in this Note or About File Type Notes must be assigned by Apple Computer, Inc. Using any file type or auxiliary type not assigned may result in conflicting identification of files by totally unrelated programs. To obtain an auxiliary type assignment in this file type, see About File Type Notes.

* Although Electronic Arts' program Music Construction Set for the Apple IIGS only creates sequences of this file type and auxiliary type $0000, the program actually attempts to read any file with type $D5. Creators of sequence files may wish to note this irregularity in their documentation.

END OF FILE FTN.D5.xxxx

```
##########################################################################
### FILE: FTN.D6.xxxx
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $D6 (214)
Auxiliary Type: All

Full Name: Instrument File
Short Name: Instrument

Written by: Matt Deatherage **January 1990**

Files of this type and auxiliary type contain musical instruments.

Files of type $D6 contain data that is to be interpreted as a definition of a musical instrument. Programs which work with music have widely varying needs; while Apple proposes a standard instrument definition (ASIF), it is largely designed for the Apple IIGS Note Synthesizer and is listed under file type $D8, Sampled Sound. Programs not using that tool may require files to store their own instruments. Apple assigns auxiliary types in this file type for such purposes.

The following auxiliary type assignments are current for this file type as of the publication date of this Note:

Auxiliary Type	Short Name	Developer
$0000	Music Construction Set inst.*	Electronic Arts
$8002	Diversi-Tune instrument	DSR

Table 1-Auxiliary Type Assignments

The auxiliary types for this file type are reserved; any not listed in this Note or About File Type Notes must be assigned by Apple Computer, Inc. Using any file type or auxiliary type not assigned may result in conflicting identification of files by totally unrelated programs. To obtain an auxiliary type assignment in this file type, see About File Type Notes.

* **Although Electronic Arts' program Music Construction Set for the Apple IIGS only creates sequences of this file type and auxiliary type $0000, the program actually attempts to read any file with type $D6. Creators of sequence files may wish to note this irregularity in their documentation.**

```
### END OF FILE FTN.D6.xxxx
```

```
##########################################################################
### FILE: FTN.D7.xxxx
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: **$D7 (215)**
Auxiliary Type: **All**

Full Name: **MIDI file**
Short Name: **MIDI data**

Written by: **Matt Deatherage** **January 1990**

Files of this type and auxiliary type contain data for a Musical Instrument Digital Interface (MIDI) peripheral.

Files of type $D7 contain data that conforms to the MIDI standard as defined by the International MIDI Association. All data contained in files of this type must follow that standard.

Although absolutely anything MIDI can fit in a standard MIDI file, developers may wish to store peripheral-specific or system-specific information in a file with a different auxiliary type for easier manipulation of the data. For example, if you were to write a program to control a Matthew IOP-1 Fabulo-Synth, you might wish to store the MIDI System Exclusive (SysEx) messages in a file with a distinctive auxiliary type. In doing so, not only could you find the information to control the Fabulo-Synth more quickly and conveniently, but other applications could also know not to even waste time presenting the file to their users for use on other peripherals. This capability is obviously not desirable to all creators of MIDI applications; however, Apple assigns auxiliary types to allow you the choice.

The following auxiliary type assignments are current for this file type as of the publication date of this Note:

Auxiliary Type	Short Name	Developer
$0000	MIDI standard data	IMA
$8001	Master Tracks Pro SysEx file	PassPort

Table 1-Auxiliary Type Assignments

The auxiliary types for this file type are reserved; any not listed in this Note or About File Type Notes must be assigned by Apple Computer, Inc. Using any file type or auxiliary type not assigned may result in conflicting identification of files by totally unrelated programs. To obtain an auxiliary type assignment in this file type, see About File Type Notes.

END OF FILE FTN.D7.xxxx

```
##########################################################################
### FILE: FTN.D8.0000
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: **$D8 (216)**
Auxiliary Type: **$0000**

Full Name: **Audio Interchange File Format File**
Short Name: **Audio IFF File**

Written by: **Matt Deatherage** **March 1989**

Files of this type and auxiliary type contain sampled sounds in Apple Computer's Audio Interchange File Format (Audio IFF).

The Audio Interchange File Format (Audio IFF) provides a standard for storing sampled sounds. The format is quite flexible, allowing the storage of monaural or multichannel sampled sounds at a variety of sample rates and sample widths.

This Note describes version 1.3 (January 4, 1989) of the Audio Interchange File Format. This Note describes Audio IFF as it pertains to Apple II developers. For a copy of the original Audio IFF specification, you may order "Audio Interchange File Format v. 1.3" from APDA.

Audio IFF conforms to the "EA IFF 85" Standard for Interchange Format Files developed by Electronic Arts.

Audio IFF is primarily an interchange format, although application designers should find it flexible enough to use as a data storage format as well. If an application does choose to use a different storage format, it should be able to convert to and from the format defined in this document. This ability to convert will facilitate the sharing of sound data between applications.

Audio IFF is the result of several meetings held with music developers over a period of ten months during 1987 and 1988. Apple Computer greatly appreciates the comments and cooperation provided by all developers who helped define this standard.

Another "EA IFF 85" sound storage format is "8SVX" IFF 8-bit Sampled Voice, by Electronic Arts. "8SVX," which handles eight-bit monaural samples, is intended mainly for storing sound for playback on personal computers. Audio IFF is intended for use with a larger variety of computers, sampled sound instruments, sound software applications, and high fidelity recording devices.

The official name for this standard is Audio Interchange File Format. If an application program needs to present the name of this format to a user, such as in a "Save As..." dialog box, the name can be abbreviated to Audio IFF. Although the Apple IIGS Sampled Instrument format is often abbreviated as "ASIF," referring to Audio IFF files by a four-letter abbreviation (i.e., "AIFF") in user-level documentation or program-generated messages should be avoided.

The Chunk Concept

The "EA IFF 85" Standard for Interchange Format Files defines an overall structure for storing data in files. Audio IFF conforms to the "EA IFF 85" standard. This Note describes those portions of "EA IFF 85" that are germane to Audio IFF. For a more complete discussion of "EA IFF 85," please refer to "EA IFF 85" Standard for Interchange Format Files.

Audio IFF, like all IFF-style storage formats, is a series of discrete pieces, or "chunks." Each chunk has an eight-byte "header," which is as follows:

ckID	4 Bytes	The ID for this chunk. These four bytes must be ASCII characters in the range $20-$7F. Spaces may not precede printing characters, although trailing spaces are allowed. Characters outside the range $20-$7F are forbidden. A program can determine how to interpret the chunk data by examining ckID.
ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID. You may think of this value as the offset to the end of the chunk. Note that this is a Reverse Long; the bytes are stored high byte first.
ckData	Chunk	The data, specific to each individual chunk. There are exactly ckSize bytes of data here. If the length of the chunk is odd, a pad byte of $00 must be added at the end. The pad byte is not included in ckSize.

Since Audio IFF is primarily an interchange format, it will come as no surprise to find that all constants, such as each chunk's ckSize field, are stored in reverse format (the bytes of multiple-byte values are stored with the high-order bytes first). This is true for all constants, which are marked in their individual descriptions by the Reverse notation.

Note: All numeric values in this Note are signed unless otherwise noted. This is different from the normal File Type Note convention.

An Audio IFF file is a collection of a number of different types of chunks. There is a Common Chunk which contains important parameters describing the sampled sound, such as its length and sample rate. There is a Sound Data Chunk which contains the actual audio samples. There are several other optional chunks which define markers, list instrument parameters, store application-specific information, etc. All of these chunks are described in detail in this Note.

File Structure

The chunks in an Audio IFF file are grouped together in a container chunk. "EA IFF 85" Standard for Interchange Format Files defines a number of container chunks, but the one used by Audio IFF is called a FORM. A FORM has the following format:

ckID	4 Bytes	The ID for this chunk. These four bytes must be "FORM."
ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID. You may think of this value as the offset to the end of the chunk. Note that this is a Reverse Long; the bytes are stored high byte first. Also note that the data portion of the chunk is broken into two parts, formType and chunks.
formType	4 Bytes	Describes what's in the FORM chunk. For Audio IFF files, formType is always "AIFF." This indicates that the chunks within the FORM pertain to sampled sound. A FORM chunk of formType AIFF is called a FORM AIFF.
chunks	Bytes	The chunks contained within the FORM. These chunks are called local chunks. A FORM AIFF along with its local chunks make up an Audio IFF file.

Figure 1 is a pictorial representation of a simple Audio IFF file. It consists of a single FORM AIFF which contains two local chunks, a Common Chunk, and a Sound Data Chunk.

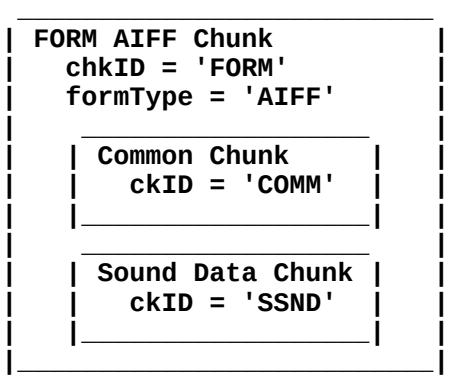


Figure 1-Simple Audio IFF File

There are no restrictions on the ordering of local chunks within a FORM AIFF.

The FORM AIFF is stored in a file with file type $D8 and auxiliary type $0000. Versions 1.2 and earlier of the Audio IFF standard used file type $CB and auxiliary type $0000. This is incorrect; the assignment listed in this Note is the correct assignment. Applications which use Audio IFF files with the older assignment should not perform adversely, since no one should be creating files of any kind with the older assignment. However, we strongly urge developers to update their applications as soon as possible to only create Audio IFF files with file type $D8 and auxiliary type $0000.

Audio IFF files may be identified in other file systems as well. On a Macintosh under MFS or HFS, the FORM AIFF is stored in the data fork of a file with file type "AIFF." This is the same as the formType of the FORM AIFF.

Note: Applications should not store any data in the resource fork of an Audio IFF file, since this information may not be preserved by all applications or in translation to foreign file systems.

Applications can use the Application Specific Chunk, described later in this Note, to store extra information specific to their application.

In file systems that use file extensions, such as MS-DOS or UNIX, it is recommended that Audio IFF file names have the extension ".AIF."

A more detailed visual example of an Audio IFF file may be found later in this Note. Please refer to it as often as necessary while reading the remainder of this Note.

Local Chunk Types

The formats of the different local chunk types found within a FORM AIFF are described in the following sections, as are their ckIDs.

There are two types of chunks: required and optional. The Common Chunk is required. The Sound Data chunk is required if the sampled sound has a length greater than zero. All other chunks are optional. All applications that use FORM AIFF must be able to read the required chunks and can choose to selectively ignore the optional chunks. A program that copies a FORM AIFF should copy all the chunks in the FORM AIFF, even those it chooses not to interpret the optional chunks.

To ensure that this standard remains usable by all developers across machine families, only Apple Computer, Inc. should define new chunk types for FORM AIFF. If you have suggestions for new chunk types, Apple is happy to listen. Please send all comments to the address listed in "About File Type Notes" to the attention of Audio IFF Suggestions.

The Common Chunk

The Common Chunk describes fundamental parameters of the sampled sound.

ckID	4 Bytes	The ID for this chunk. These four bytes must be "COMM."
ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID. For the Common Chunk, this is always 18.
numChannels	Rev. Word	The number of audio channels for the sound. A value of 1 means monophonic sound, 2 means stereo, 4 means four-channel sound, and so on. Any number of audio channels may be represented. The actual sounds samples are stored in the Sound Data Chunk.
numSampleFrames	Rev. Unsigned Long	The number of sample frames in the Sound Data Chunk. Sample frames are described below. Note that numSampleFrames is the number of sample frames, not the number of bytes nor the number of sample points (also described below) in the Sound Data Chunk. The total number of sample points in the file is numSampleFrames multiplied by numChannels.
sampleSize	Rev. Word	The number of bits in each sample point. This can be any number from 1 to 32.

sampleRate Rev. Extended	The sample Rate at which the sound is to be played back, in sample frames per second.

One, and only one, Common Chunk is required in every FORM AIFF.

Sample Points and Sample Frames

A large part of interpreting Audio IFF files revolves around the two concepts of sample points and sample frames.

A sample point is a value representing a sample of a sound at a given point in time. A sample point may be from 1 to 32 bits wide, as determined by sampleSize in the Common Chunk. Sample points are stored in an integral number of contiguous bytes. One- to eight-bit wide sample points are stored in one byte, 9- to 16-bit wide sample points are stored in two bytes, 17- to 24-bit wide sample points are stored in three bytes, and 25- to 32-bit wide sample points are stored in four bytes (most significant byte first). When the width of a sample point is not a multiple of eight bits, the sample point data is left justified, with the remaining bits zeroed. An example case is illustrated in Figure 2. A 12-bit sample point, binary 101000010111, is stored left justified in two bytes. The remaining bits are set to zero.

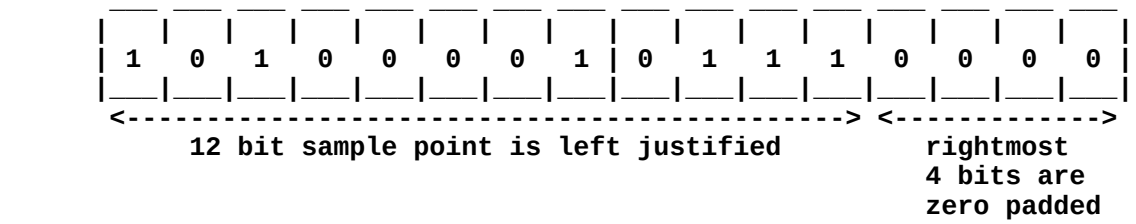


Figure 2-A 12-Bit Sample Point

Sample frames are sets of sample points which are interleaved for multichannel sound. Single sample points from each channel are interleaved such that each sample frame is a sample point from the same moment in time for each channel available. This is illustrated in Figure 3 for the stereo (two channel) case.

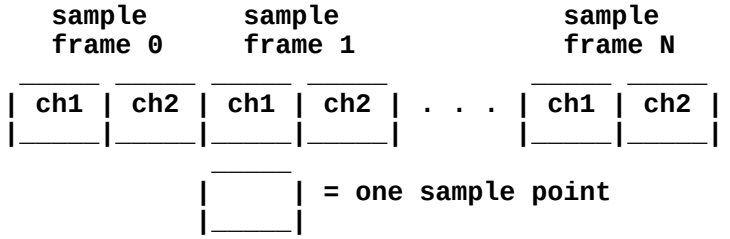
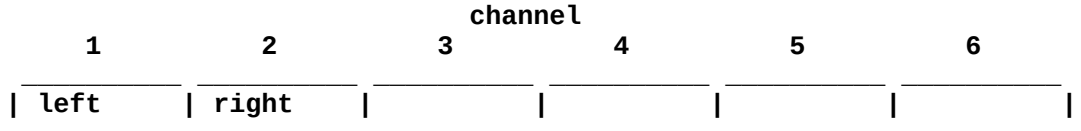


Figure 3-Sample Frames for Multichannel Sound

For monophonic sound, a sample frame is a single sample point. For multichannel sounds, you should follow the conventions in Figure 4.



stereo	left	right				
3 channel	left	right	center			
quad	front left	front right	rear left	rear right		
4 channel	left	center	right	surround		
6 channel	left	left center	center	right	right center	surround

Figure 4-Sample Frame Conventions for Multichannel Sound

Note: Portions of Figure 4 do not follow the Apple IIGS standard of right on even channels and left on odd channels. The portions that do follow this convention usually use channel two for right instead of channel zero as most Apple IIGS standards. Be prepared to interpret data accordingly.

Sample frames are stored contiguously in order of increasing time. The sample points within a sample frame are packed together; there are no unused bytes between them. Likewise, the sample frames are packed together with no pad bytes.

The Sound Data Chunk

The Sound Data Chunk contains the actual sample frames.

ckID	4 Bytes	The ID for this chunk. These four bytes must be "SSND."
ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID.
offset	Rev. Unsigned Long	Determines where the first sample frame in the soundData starts, in bytes. Most applications will not use offset and should set it zero. Use for a non-zero offset is explained below.
blockSize	Rev. Unsigned Long	Used in conjunction with offset for block-aligning sound data. It contains the size in bytes of the blocks to which soundData is aligned. As with offset, most applications will not use blockSize and should set it to zero. More information on blockSize is given below.
soundData	Bytes	Contains the actual sample frames that make up the sound. The number of sample frames in the soundData is determined by the numSampleFrames parameter in the Common Chunk.

The Sound Data Chunk is required unless the numSampleFrames field in the Common Chunk is zero. A maximum of one Sound Data Chunk may appear in a FORM AIFF.

Block-Aligning Sound Data

There may be some applications that, to ensure real time recording and playback of audio, wish to align sampled sound data with fixed-size blocks. This alignment can be accomplished with the offset and blockSize parameters of the Sound Data Chunk, as shown in Figure 5.

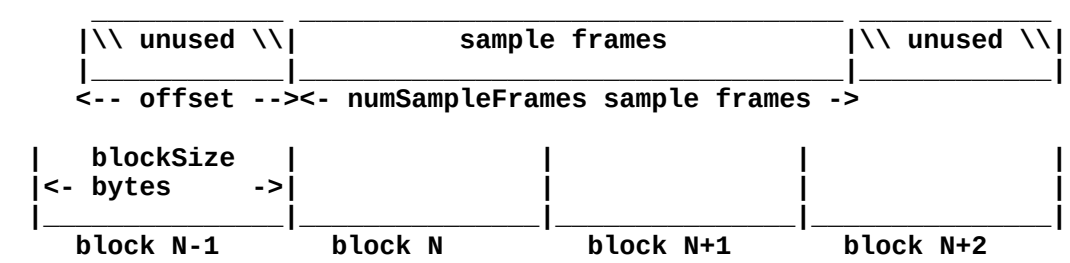


Figure 5-Block-Aligned Sound Data

In Figure 5, the first sample frame starts at the beginning of block N. This is accomplished by skipping the first offset bytes of the soundData. Note too, that the soundData bytes can extend beyond valid sample frames, allowing the soundData bytes to end on a block boundary as well.

The blockSize specifies the size in bytes of the block to which you would align the sound data. A blockSize of zero indicates that the sound data does not need to be block-aligned. Applications that don't care about block alignment should set the blockSize and offset to zero when creating Audio IFF files. Applications that write block-aligned sound data should set blockSize to the appropriate block size. Applications that modify an existing Audio IFF file should try to preserve alignment of the sound data, although this is not required. If an application does not preserve alignment, it should set the blockSize and offset to zero. If an application needs to realign sound data to a different sized block, it should update blockSize and offset accordingly.

The Marker Chunk

The Marker Chunk contains markers that point to positions in the sound data. Markers can be used for whatever purposes an application desires. The Instrument Chunk, defined later in this Note, uses markers to mark loop beginning and end points.

ckID	4 Bytes	The ID for this chunk. These four bytes must be "MARK."
ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID.
numMarkers	Rev. Unsigned Word	The number of markers (defined below) in the Marker Chunk. If non-zero, this is followed by the markers themselves. Because all fields in a marker are an even number of bytes, the length of any marker will always be even. Thus, markers are packed together with no unused bytes between them, although the markers themselves need not be ordered in any particular manner.

Marker	Markers	Defined below.

A marker has the following format:

MarkerID	Rev. Word	The ID for this marker. This is a number that uniquely identifies the marker within a FORM AIFF. The number can be any positive, non-zero integer, as long as no other marker within the same FORM AIFF has the same ID.
position	Rev. Unsigned Long	Determines the marker's position in the sound data. Markers conceptually fall between two sample frames. A marker that falls before the first sample frame in the sound data is at position zero, while a marker that falls between the first and second sample frame in the sound data is at position one. Units for position are sample frames, not bytes nor sample points.
markerName	String	Pascal-type string containing the name of the mark.

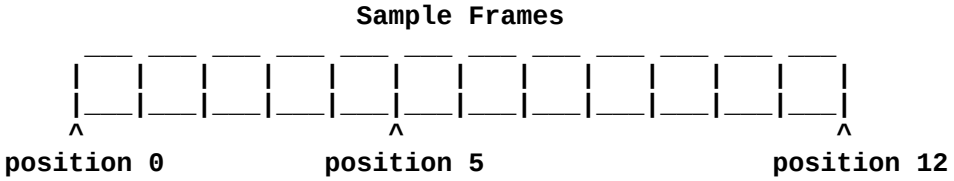


Figure 6-Sample Frame Marker Positions

Note: Some "EA IFF 85" files store strings as C-style strings (null terminated). Audio IFF uses Pascal-style (length byte) strings because they are easier to skip over when scanning a file or a chunk.

The Marker Chunk is optional. No more than one Marker Chunk can appear in a FORM AIFF.

The Instrument Chunk

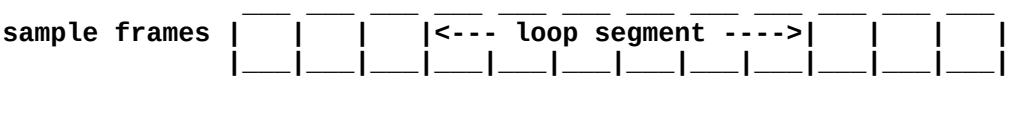
The Instrument Chunk defines basic parameters that an instrument, such as a sample, could use to play the sound data.

ckID	4 Bytes	The ID for this chunk. These four bytes must be "INST."
ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID. For the Instrument Chunk, this field is always 20.
baseNote	Byte	The note at which the instrument plays the sound data without pitch modification. Units are MIDI (Musical Instrument Digital Interface) note numbers, and are in the range 0 through 127. Middle C is 60.
detune	Byte	Determines how much the instrument should alter the

		pitch of the sound when it is played . Units are cents (1/100 of a semitone), and range from -50 to +50. Negative numbers mean that the pitch of the sound should be lowered, while positive numbers mean that it should be raised.
lowNote	Byte	Suggested lowest note on a keyboard for playback of the sound data. The sound data should be played if the instrument is requested to play a note between the lowNote and highNote, inclusive. The base note does not have to be within this range. Units for lowNote and highNote are MIDI note values.
highNote	Byte	Suggested highest note on a keyboard for playback of the sound data. See the description of lowNote above.
lowVelocity	Byte	The low end of the suggested range of velocities for playback of the sound data. The sound data should be played if the note-on velocity is between lowVelocity and highVelocity, inclusive. Units are MIDI velocity values, 1 (lowest velocity) through 127 (highest velocity).
highVelocity	Byte	The high end of the suggested range of velocities for playback of the sound data. See the description of lowVelocity above.
gain	Rev. Word	The amount by which to change the gain of the sound when it is played. Units are decibels. For example, 0 dB means no change, 6 dB means double the value of each sample point, while -6 dB means halve the value of each sample point.
sustainLoop	Loop	A loop that is to be played when an instrument is sustaining a sound. The format of loops is described below.
releaseLoop	Loop	A loop that is to be played when an instrument is in the release phase of playing back a sound. The release phase usually occurs after a key on an instrument is released. The format of loops is described below.

Loops

Sound data can be looped, allowing a portion of the sound to be repeated to lengthen the sound. A loop is marked with two points, a begin position and an end position. There are two ways to play a loop, forward looping and forward/backward looping. In the case of forward looping, playback begins at the beginning of the sound, continues past the begin position and continues to the end position, at which point playback starts again at the begin position. The segment between the begin and end positions, called the loop segment, is played repeatedly until interrupted by a user action, such as the release of a key on a sampling instrument.



```
                    ^                    ^
              begin position       end position
```

Figure 7-Sample Frame Looping

With forward/backward looping, the loop segment is first played from the begin position to the end position, and then played backwards from the end position to the begin position. This flip-flop pattern is repeated over and over again until interrupted.

The following structure describes a loop:

playMode	Rev. Word	The type of looping to be performed. 0 = no looping 1 = Forward looping 2 = Forward/Backward looping If 0 is specified, the loop points are ignored during playback.
beginLoop	Rev. Word	A Marker ID of the marker to the begin position.
endLoop	Rev. Word	A Marker ID of the marker to the end position. The begin position must be less than the end position. If this is not the case, the loop segment has zero or negative length and no looping occurs.

The Instrument Chunk is optional. No more than one Instrument Chunk can appear in a FORM AIFF.

ASIF Note: The Apple IIGS Sampled Instrument Format also defines a chunk with ID of "INST," which is not the same as the Audio IFF Instrument Chunk. A good way to tell the two chunks apart in generic IFF-style readers is by the ckSize fields. The Audio IFF Instrument Chunk's ckSize field is always 20, whereas the Apple IIGS Sampled Instrument Format Instrument Chunk's ckSize field, for structural reasons, can never be 20.

The MIDI Data Chunk

The MIDI Data Chunk can be used to store MIDI data. Please refer to Musical Instrument Digital Interface Specification 1.0, available from the International MIDI Association, for more details on MIDI.

The primary purpose of this chunk is to store MIDI System Exclusive messages, although other types of MIDI data can be stored in the block as well. As more instruments come to market, they will likely have parameters that have not been included in the Audio IFF specification. The MIDI System Exclusive messages for these instruments may contain many parameters that are not included in the Instrument Chunk. For example, a new sampling instrument may have more than the two loops defined in the Instrument Chunk. These loops will likely be represented in the MIDI System Exclusive message for the new machine. This MIDI System Exclusive message can be stored in the MIDI Data Chunk.

ckID	4 Bytes	The ID for this chunk. These four bytes must be "MIDI."

ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID.
MIDIdata Unsigned Bytes		A stream of MIDI Data.

The MIDI Data Chunk is optional. Any number of MIDI Data Chunks may exist in a FORM AIFF. If MIDI System Exclusive messages for several instruments are to be stored in a FORM AIFF, it is better to use one MIDI Data Chunk per instrument than one big MIDI Data Chunk for all of the instruments.

The Audio Recording Chunk

The Audio Recording Chunk contains information pertinent to audio recording devices.

ckID	4 Bytes	The ID for this chunk. These four bytes must be "AESD."
ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID. For the Audio Recording Chunk, this value is always 24.
AESChannelStatusData 24 Bytes		These 24 bytes are specified in the AES Recommended Practice for Digital Audio Engineering--Serial Transmission Format for Linearly Represented Digital Audio Data, section 7.1, Channel Status Data. This document describes a format for real-time digital transmission of digital audio between audio devices. This information is duplicated in the Audio Recording Chunk for convenience. Bits 2, 3, and 4 of byte zero are of general interest as they describe recording emphasis.

The Audio Recording Chunk is optional. No more than one Audio Recording Chunk may appear in a FORM AIFF.

The Application Specific Chunk

The Application Specific Chunk can be used for any purposes whatsoever by developers and application authors. For example, an application that edits sounds might want to use this chunk to store editor state parameters such as magnification levels, last cursor position, etc.

ckID	4 Bytes	The ID for this chunk. These four bytes must be "APPL."
ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID. For the Audio Recording Chunk, this value is always 24.
OSType	4 Bytes	Identifies a particular application. For Apple II applications, these four bytes should always be 'pdos' ($70 $64 $6F $73). In this case, the beginning of the data area is defined to be a Pascal string containing the name of the application. For Macintosh applications, this is simply

		the four-character signature as registered with Developer Technical Support.
AppSignature	String	Pascal string identifying the application.
data	Bytes	Data specific to the application.

Note: AppSignature does not exist unless OSType is "pdos." In all other cases, the data area starts immediately following the OSType field.

The Application Specific Chunk is optional. Any number of Application Specific Chunks may exist in a single FORM AIFF.

The Comments Chunk

The Comments Chunk is used to store comments in the FORM AIFF. "EA IFF 85" has an Annotation Chunk (used in ASIF) that can be used for comments, but the Comments Chunk has two features not found in the "EA IFF 85" chunk. They are a time-stamp for the comment and a link to a marker.

ckID	4 Bytes	The ID for this chunk. These four bytes must be "COMT."
ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID.
numComments	Rev. Unsigned Word	The number of comments in the Comments Chunk. This is followed by the comments themselves. Comments are always an even number of bytes in length, so there is no padding between comments in the Comments Chunk.
Comment	Comment	The comments. There are numComments of them.

The format of a comment is described below:

timeStamp	Rev. Unsigned Long	Indicates when the comment was created. Units are the number of seconds since 12:00 a.m. (midnight), January 1, 1904. This is the standard Macintosh time format. Macintosh routines to manipulate this time stamp may be found in Inside Macintosh, Volume II.

Note: The routine to convert timeStamp into a standard GS/OS date, as described in the Audio IFF 1.3 specification, is not available at this time.

marker	Rev. Word	A Marker ID. If this comment is linked to a marker (to store a long description of a marker as a comment, for example), this is the ID of that marker. Otherwise marker is zero, indicating there is no such link.
count	Rev. Word	Count of the number of characters in the following text. By using a word instead of a byte, much larger comments may be created.
text	Bytes	The comment itself. If the text is an odd number of bytes in length, it must be padded with a zero byte to ensure that it is an even

		number of bytes in length. If the pad byte is present, it is not included in count.

The Comments Chunk is optional. No more than one Comments Chunk may appear in a single FORM AIFF.

The Text Chunks

These four chunks are included in the definition of every "EA IFF 85" file. All are text chunks; their data portion consists solely of text. Each of these chunks is optional.

The Name Chunk

This chunk names the sampled sound.

ckID	4 Bytes	The ID for this chunk. These four bytes must be "NAME."
ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID.
Name	Bytes	ASCII characters ($20-$7F) representing the name. There should be ckSize characters.

No more than one Name Chunk may exist within a FORM AIFF.

The Author Chunk

This chunk can be used to identify the creator of a sampled sound.

ckID	4 Bytes	The ID for this chunk. These four bytes must be "AUTH."
ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID.
author	Bytes	ASCII characters ($20-$7F) representing the name of the author of the sampled sound. There should be ckSize characters.

No more than one Author Chunk may exist within a FORM AIFF.

The Copyright Chunk

The Copyright Chunk contains a copyright notice for the sound. The copyright contains a date followed by the copyright owner. The chunk ID "(c) " serves as the copyright character ((C)). For example, a Copyright Chunk containing the text "1989 Apple Computer, Inc." means "(C) 1989 Apple Computer, Inc."

ckID	4 Bytes	The ID for this chunk. These four bytes must be "(c) ."
ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID. You may think of this value as the offset to the end of the chunk.
notice	Bytes	ASCII characters ($20-$7F) representing a copyright notice for the voice or collection of voices. There should be ckSize characters.

No more than one Copyright Chunk may exist within a FORM AIFF.

The Annotation Chunk

Use of this comment is discouraged within FORM AIFF. The more powerful Comments Chunk should be used instead.

ckID	4 Bytes	The ID for this chunk. These four bytes must be "ANNO."
ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID. You may think of this value as the offset to the end of the chunk. Note that this is a Reverse Long; the bytes are stored high byte first.
author	Bytes	ASCII characters ($20-$7F) representing the name of the author of the voices or collection of voices. There should be ckSize characters.

Many Annotation Chunks may exist within a FORM AIFF.

Chunk Precedence

Several of the local chunks for FORM AIFF may contain duplicate information. For example, the Instrument Chunk defines loop points and MIDI System Exclusive data in the MIDI Data Chunk may also define loop points. What happens if these loop points are different? How is an application supposed to loop the sound? Such conflicts are resolved by defining a precedence for chunks. This precedence is illustrated in Figure 8.

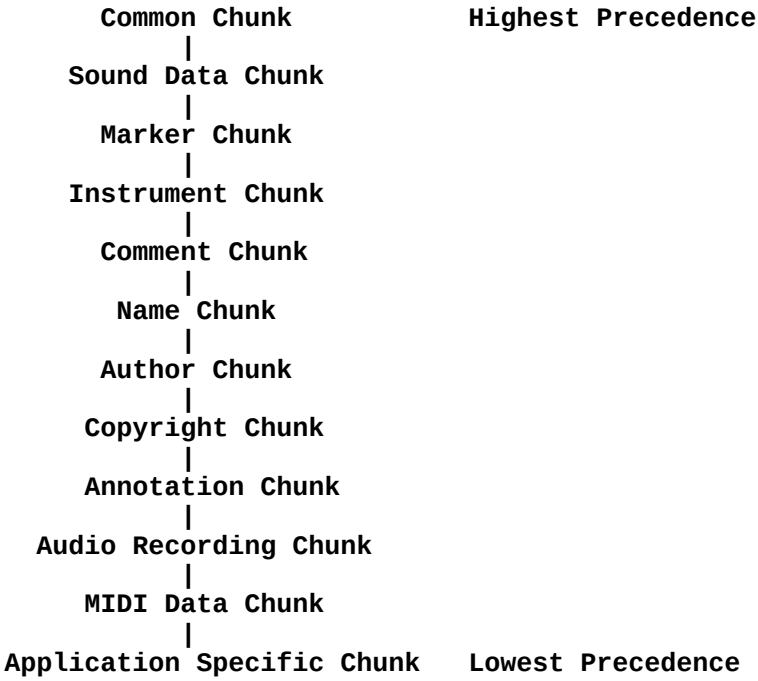


Figure 8-Chunk Precedence

The Common Chunk has the highest precedence, while the Application Specific

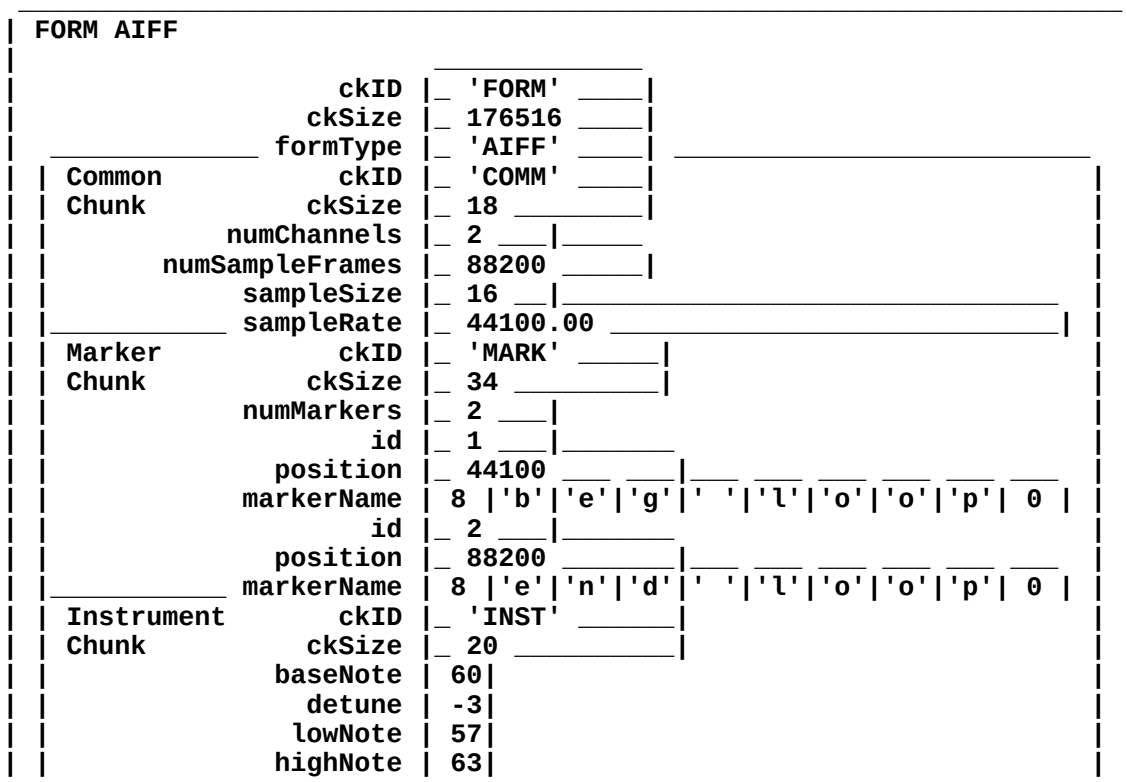
Chunk has the lowest. Information in the Common Chunk always takes precedence over conflicting information in any other chunk. The Application Specific Chunk always loses in conflicts with other chunks. By looking at the chunk hierarchy, for example, one sees that the loop points in the Instrument Chunk take precedence over conflicting loop points found in the MIDI Data Chunk.

It is the responsibility of applications that write data into the lower precedence chunks to make sure that the higher precedence chunks are updated accordingly.

Figure 9 (on the following page) illustrates an example of a FORM AIFF. An Audio IFF file is simple a file containing a single FORM AIFF. The FORM AIFF is stored in the data fork of file systems that can handle resource forks.

Further Reference

- Inside Macintosh, Volume II
- Apple Numerics Manual, Second Edition
- File Type Note File Type $D8, Auxiliary Type $0002, Apple IIGS Sampled Instrument Format
- Audio Interchange File Format v1.3 (APDA)
- AES Recommended Practice for Digital Audio Engineering--Serial Transmission Format for Linearly Represented Digital Audio Data, Audio Engineering Society, 60 East 42nd Street, New York, NY 10165
- MIDI: Musical Instrument Digital Interface, Specification 1.0, the International MIDI Association.
- "EA IFF 85" Standard for Interchange Format Files (Electronic Arts)
- "8SVX" IFF 8-bit Sampled Voice (Electronic Arts)



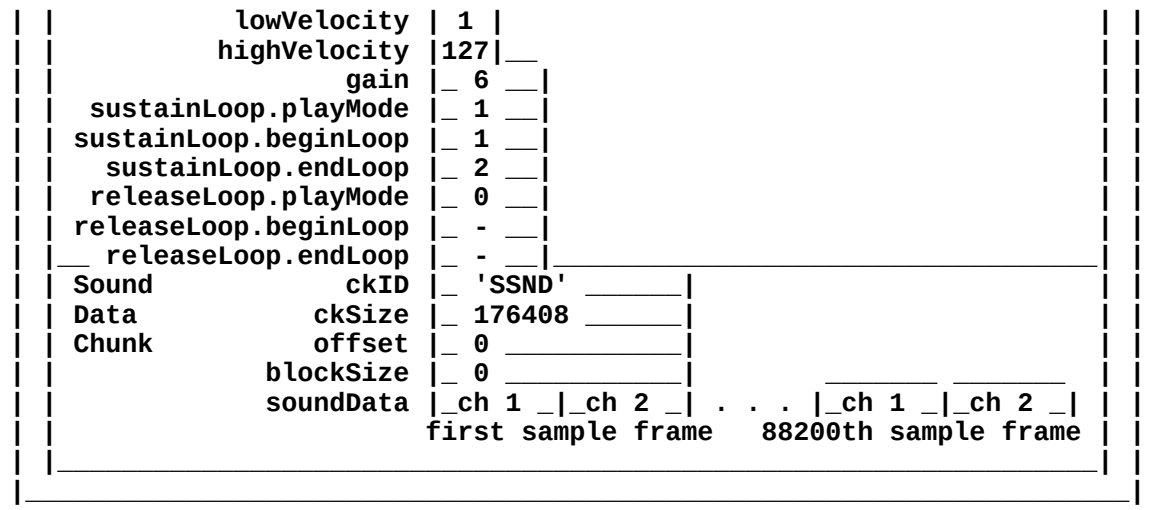


Figure 9-Sample FORM AIFF

END OF FILE FTN.D8.0000

```
##########################################################################
### FILE: FTN.D8.0002
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: **$D8 (216)**
Auxiliary Type: **$0002**

Full Name: **Apple IIGS Sampled Instrument File**
Short Name: **ASIF File**

Written by: **John Worthington & Matt Deatherage** **March 1989**

Files of this type and auxiliary type contain instruments in the Apple IIGS Sampled Instrument Format (ASIF).

The Apple IIGS Sampled Instrument Format (ASIF) is the standard format used for storing Apple IIGS Sampled Instruments on disk. All sampled instruments supplied by Apple for the Apple IIGS Note Synthesizer will use this format. Likewise, all utilities supporting Note Synthesizer sampled instruments will also use this format.

ASIF is designed around the needs of the current Apple IIGS Note Synthesizer and Apple IIGS sound hardware. While the format of ASIF (especially INST chunks) is greatly influenced by the Note Synthesizer, the information may be sufficient for other sampled sound synthesizers to accurately render the sound.

Most instrument files for the Apple IIGS have a ProDOS file type of $D6. ASIF files are instead identified as file type $D8, auxiliary type $0002 because of their sampled nature. All other instruments for the the Apple IIGS will be identified by file type $D6.

Note: Earlier ASIF documentation, not widely circulated, defined ASIF files as file type $CA, auxiliary type $8000. As documented in this Note, ASIF has been reassigned to file type $D8 and auxiliary type $0002. Applications which read files based on the old file type and auxiliary type should not perform adversely, since no other application should be creating files with that combination. However, we strongly urge developers to create ASIF files with file type $D8 and auxiliary type $0002 only. We further encourage developers to revise existing programs to use this new assignment at their earliest convenience.

ASIF files conform to the "EA IFF 85" Standard for Interchange Format Files developed by Electronic Arts. Electronic Arts additionally has some public domain code available for reading and writing IFF files.

ASIF is provided more for compatibility than for interchange. It is a highly Apple IIGS specific file format. Those wishing the highest level of sampled sound compatibility across programs and CPUs should use Audio Interchange File Format (Audio IFF). Audio IFF is documented in Apple II File Type Note for file type $D8, auxiliary type $0000.

This Note defines the required chunks INST and WAVE, as well as the optional ("NAME"), copyright ("(c) "), author ("AUTH"), and annotation ("ANNO") chunks. These are all "standard" chunks. Additional chunks for private or future needs may be added later. Figure 1, located at the end of this Note, illustrates the ASIF format in a box diagram.

Required Data Chunks

An ASIF file consists of a single FORM ASIF, which contains one and only one WAVE chunk and one or more INST chunks. Each ASIF file defines at least one instrument.

INST chunks contain all of the Note Synthesizer specific information needed to define an instrument, exclusive of the actual wave form. The information in the INST chunk defines the characteristics of an instrument such as the envelope, pitch range, and maximum pitch bend. There must be at least one INST chunk for each instrument in the ASIF file.

WAVE chunks contain the waveforms for a given instrument. A WAVE chunk may contain waveforms used by more than one instrument. In most cases, the waveforms used by an application will be merged into a single 64K block that is loaded into DOC RAM when the application is launched. In this case, there would be several INST chunks referring to that single WAVE chunk. Most music applications will probably store instruments one to a file, which is the preferred way of distributing ASIF instruments.

Note: The length of any chunk must be even. If a chunk has an odd length, a pad byte of $00 must be added to the end of the chunk. The pad byte, if present, should never have a value other than $00.

The FORM Chunk

ckID	4 Bytes	The ID for this chunk. These four bytes must be "FORM."
ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID. You may think of this value as the offset to the end of the chunk. Note that this is a Reverse Long; the bytes are stored high byte first.
ckType (chunks...)	4 Types	The type of chunk. These four bytes must be "ASIF."

Immediately following the 12-byte FORM chunk header are the data chunks of the ASIF file. There most be one and only one WAVE chunk, and at least one INST chunk. Optionally there may be name ("NAME"), copyright ("(c) "), author ("AUTH"), or annotation ("ANNO") chunks. All data chunks are part of the larger FORM chunk, referred to as the FORM ASIF because of the ID and Type of this chunk.

The INST Chunk

ckID	4 Bytes	The ID for this chunk. These four bytes must be "INST."

ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID. You may think of this value as the offset to the end of the chunk. Note that this is a Reverse Long; the bytes are stored high byte first.
InstName	String	A Pascal String containing the name of the instrument referred to by this INST block. This string should be used as the display name of the instrument.

Note: The length byte of InstName is also referred to as INameLength.

SampleNum	Word	The number of the sample in the WAVE chunk to which this instrument refers.
Envelope	8 InstSegs	Eight linear InstSegs defining the instrument's envelope.

The InstSeg is a three-byte linear segment that describes a level and a slope. The level is called the breakpoint and represents the linear amplitude of the sound. The slope is described by an increment added or subtracted from the current level at the update rate. Regardless of the increment, the breakpoint will never be exceeded. All ASIF instruments assume an update rate of 200 Hz. The increment is a two-byte fixed pointer; that is, the lower eight bits represent a fraction. Thus when the increment is one, it represents 1/256. In this case, the increment would have to be added 256 times (1.28 seconds) to cause the level to go up by 1. At a 200 Hz update rate each increment takes 5 milliseconds. If an application wishes to use an update rate other than 200 Hz, the envelope must be scaled as necessary. If the envelope is not scaled, the instrument will not sound correct.

The breakpoint is a byte between 0 and 127 ($00 and $7F). It should represent sound level in a logarithmic scale: every 16 steps change the amplitude by 6 dB.

Therefore the envelope is composed of eight InstSegs:

stage1	Byte	Breakpoint 1
	2 Bytes	Increment 1
stage2	Byte	Breakpoint 2
	2 Bytes	Increment 2
stage3	Byte	Breakpoint 3
	2 Bytes	Increment 3
stage4	Byte	Breakpoint 4
	2 Bytes	Increment 4
stage5	Byte	Breakpoint 5
	2 Bytes	Increment 5
stage6	Byte	Breakpoint 6
	2 Bytes	Increment 6
stage7	Byte	Breakpoint 7
	2 Bytes	Increment 7
stage8	Byte	Breakpoint 8
	2 Bytes	Increment 8

Increment 1 is used to go from the initial level of 0 up to the level of breakpoint 1. Increment 2 is used to go from breakpoint 1 to breakpoint 2, and so on. The sustain level of the envelope, if there is one, is created by setting the increment to zero, causing the envelope to get

stuck on that level. The last segment used for release should always have a breakpoint of zero, so the sound eventually reaches silence. Unused segments should have a zero breakpoint and a non-zero increment.

ReleaseSegment	Byte	Specifies the release segment of the envelope. This must be a number from 1 to 7. The release may take several segments to get to zero. The last segment should always be zero.
PriorityIncrement	Byte	A number that will be subtracted from the generator priority when the envelope reaches the sustain segment. The sustain segment is the first segment with a zero increment. When the release segment is reached, the priority is cut in half. The priority of each generator is also decremented by one each time a new generator is allocated. This causes older notes to be preferred for stealing.
PitchBendRange	Byte	The number of semitones that the pitch will be raised when the pitchwheel reaches 127 (the center value is 64). The legal values for PitchBendRange are 1, 2, and 4.
VibratoDepth	Byte	The initial fixed depth of vibrato, ranging from 0 to 127. Vibrato is a triangular-shaped Low Frequency Oscillator (LFO) modulating the pitch of both oscillators in a generator. A VibratoDepth of zero turns the vibrator mechanism off, which saves some CPU time (since vibrato is implemented in software).
VibratoSpeed	Byte	Controls the rate of the vibrato LFO. It can be any byte value, although the range from 5 to 20 is most useful. The frequency range is linear, in 0.2 Hz steps.
UpdateRate	Byte	Unused; set to zero. Previous versions of ASIF listed this byte as the update rate in .4 Hz, but a one-byte field is not large enough to provide suitable resolution (102 Hz is the maximum allowed), much less the standard Note Synthesizer value of 200 Hz (the byte would have to hold the value 500; not an easy task for a byte). All ASIF instruments are assumed to have an update rate of 200 Hz.
AWaveCount	Byte	The number of waves in the following AWaveList. There can be up to 255 waves in the AWaveList.
BWaveCount	Byte	The number of waves in the following BWaveList. There can be up to 255 waves in the BWaveList.
AWaveList	AWaveCount	Waves.
BWaveList	BWaveCount	Waves.

The WaveList structure is a variable-length array where each entry is six bytes long. The information is particular to the DOC, and the developer should refer to the DOC information in the Apple IIGS Hardware Reference and the Apple IIGS Toolbox Reference Update when creating

instruments. Each six-byte entry represents a waveform and contains information about the allowable pitch range of the waveform. This means that the waves can be "multi-sampled" across an imaginary keyboard. When a note is played, WaveListA and WaveListB will be examined, and one waveform will be picked and assigned to each oscillator.

Each wave in a WaveList has the following 6-byte format:

TopKey	Byte	The highest MIDI semitone this waveform will play. The Note Synthesizer will examine the TopKey field of each waveform until it finds one greater than or equal to the note it is trying to play. The items in the WaveList should be in order of increasing TopKey values. The last wave should have a TopKey value of 127. The TopKey value is the split point between the waveforms.

The next three bytes will be stuffed into the DOC registers:

WaveAddress	Byte	The high byte of the waveform address. Note that the value selected for WaveAddress should assume that the waveform starts in page zero. When the waveform is actually placed in DOC RAM, the values must be adjusted as appropriate. As an example, for a waveform starting at $8000 in DOC RAM, this value would be $80.
WaveSize	Byte	Sets both the size of the wavetable and the frequency resolution.
DOCMode	Byte	Placed in the DOCs Mode register. The interrupt-enable should always be zero. Some ways this may be used are: Synthesizer ($00), where both oscillators (A and B) run in free run mode Sample, no loop: Oscillator A in swap mode ($06) and oscillator B in one-shot halted mode ($03). Oscillator A will play its wave once and start Oscillator B, which will play its wave to the end once and stop. Sampled with loop: Oscillator A in swap mode ($06), and Oscillator B in free-run halted mode ($01). Oscillator A will play its wave once and then start Oscillator B, which will play continuously until the note ends. The high nibble of the DOCMode is the channel number. This must be set correctly for stereo output. While all of the currently available stereo cards will map even-numbered channels to the right and odd-numbered channels to the left, software should use channel 0 for right and channel 1 for left. This will ensure compatibility with cards that

		provide more than two channels of output. If you are not designing stereo instruments, always set the channel to zero.
RelPitch	Word	Used to tune the waveform. This will compensate for different sample rates and waveform sizes. The high byte is in semitones, but can be a signed number. The low byte is in 1/256 semitone increments. Note that the low byte is first in memory; this is a regular 65816 Word. A setting of zero is the default for sounds that gave one cycle per page of waveform memory.

The WaveList structure is designed to give greater flexibility in creating realistic instrumental timbres. It allows "multi-sampling" with different samples of sounds on different ranges of pitch. It allows mixing of various sized wave forms, with different tuning on each semitone, to allow separate tuning of each note. This is one way to duplicate special tuning systems like "just temperament." The wave pointers need not be different in this case, just the RelPitch fields.

Tuning is accurate to 1/128 of a semitone in the Note Synthesizer, subject to the resolution setting of the DOC. For accurate tuning on lower notes, it may be necessary to use higher settings in the DOC resolution register.

Note: The Audio Interchange File Format (Audio IFF) also has a chunk named "INST" which will appear to a standard IFF reader the same as the ASIF "INST" chunk. To tell the two apart, check the ckSize field. The Audio IFF "INST" chunk will always have ckSize of 20 bytes, and the ASIF "INST" chunk will never have a chunk size of 20 bytes.

The WAVE chunk

ckID	4 Bytes	The ID for this chunk. These four bytes must be "WAVE."
ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID. You may think of this value as the offset to the end of the chunk. Note that this is a Reverse Long; the bytes are stored high byte first.
WaveName	String	A Pascal String containing the name of the waveform referred to by this WAVE block.

Note: The length byte of WaveName is also referred to as WaveNameLen.

WaveSize	Word	The size of the waveform WaveData, in bytes. WaveSize may be any value from $0000 to $FFFF. This is a zero-based counter; WaveData that is one byte long would result in a WaveSize of $0000. This allows full 64K WaveData entries.
NumSamples	Word	The number of different sounds in this WAVE chunk. NumSamples describes the number of entries in SampleTable. Note that this is

		not necessarily the number of instruments. Although not required, there should be a WaveList entry in an INST chunk for each entry in the SampleTable.
SampleTable	NumSamples	Samples.

SampleTable is a table of the waveforms corresponding to different "samples". Each entry in SampleTable is 12 bytes long. Each sample entry is defined as follows:

Location	Word	The byte offset to the waveform from the beginning of the WAVE chunk.
Size	Word	The size of the waveform in 256-byte pages. Size is specified in pages since the sample size passed to the DOC must be in pages.
OrigFreq	Fixed	The original frequency that was sampled, in hertz. For example, if A440 was sampled, the value of this field would be 440.00. A value of zero in this field means that the original frequency of the sample is unknown.
SampRate	Fixed	The sample rate used to generate this sample, in hertz. A value of zero in this field means that the original sample rate is unknown.

There are NumSamples of these sample entries in the SampleTable.

WaveData	WaveSize Bytes	The actual waveform. The DOC uses samples in an eight-bit linear format. A value of $80 is considered to be a zero crossing. Positive values are greater than $80; negative values are less than $80. Although WaveData may contain zeros as oscillator control values, it should never contain a zero value as a sample value since this halts the oscillator.

Optional Data Chunks

There are currently three types of optional data chunks. These chunks may be included in an ASIF file if desired. They are considered part of the set of "standard" chunks in the Electronic Arts "EA IFF 85" definition.

The NAME Chunk

This chunk names the instrument of collection of instruments defined in the ASIF file. This chunk may be used to supply a display name for a collection of instruments. This can be useful since IFF programs know about the NAME chunk, but may not know about the name field in INST or WAVE chunks.

ckID	4 Bytes	The ID for this chunk. These four bytes must be "NAME."
ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID.
Name	Bytes	ASCII characters ($20-$7F) representing the name.

		There should be ckSize characters.

The AUTH Chunk

ckID	4 Bytes	The ID for this chunk. These four bytes must be "AUTH."
ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID.
author	Bytes	ASCII characters ($20-$7F) representing the name of the author of the voices or collection of voices. There should be ckSize characters.

The "(c) " Chunk

ckID	4 Bytes	The ID for this chunk. These four bytes must be "(c) ."
ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID. You may think of this value as the offset to the end of the chunk.
notice	Bytes	ASCII characters ($20-$7F) representing a copyright notice for the voice or collection of voices. There should be ckSize characters.

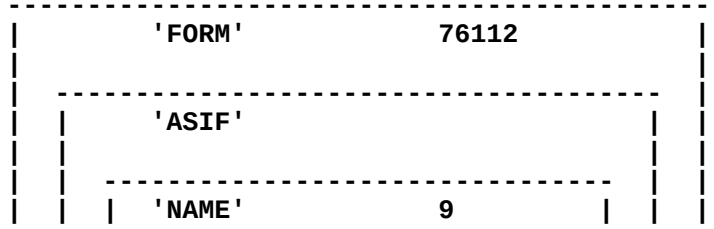
The ANNO Chunk

ckID	4 Bytes	The ID for this chunk. These four bytes must be "ANNO."
ckSize	Rev. Long	The length of this chunk, excluding ckSize and cdID. You may think of this value as the offset to the end of the chunk. Note that this is a Reverse Long; the bytes are stored high byte first.
author	Bytes	ASCII characters ($20-$7F) representing the name of the author of the voices or collection of voices. There should be ckSize characters.

Other Chunk Types

There are many types of IFF chunks other than those described in this document. New chunks may be added to ASIF files in the future. If an application encounters a chunk it doesn't recognize when reading an ASIF file, it should ignore it. Note that all chunks should be preserved when copying an IFF file.

Figure 1 illustrates a sample ASIF file as a box diagram.



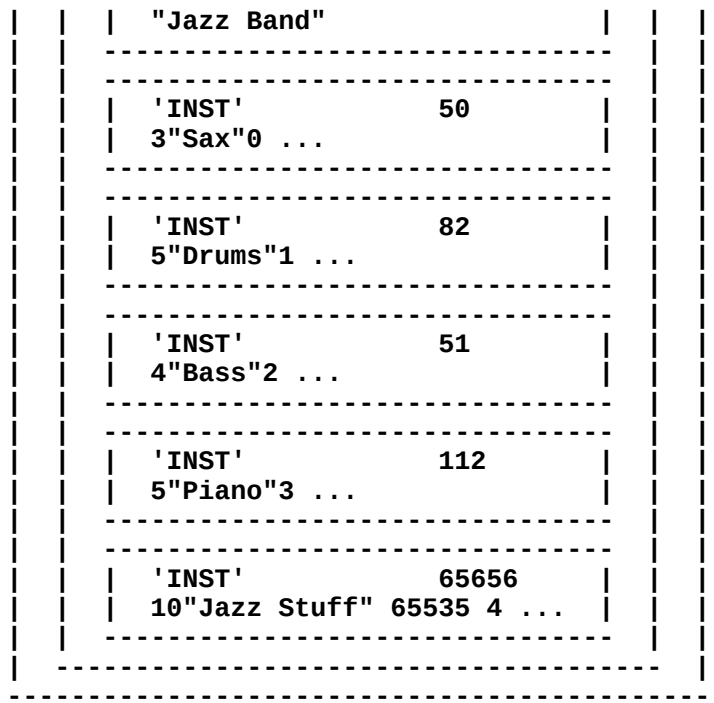


Figure 1-Sample ASIF File

Further Reference

- Apple IIGS Toolbox Reference Update
- Advanced Sampler's Guide (Ensoniq Corporation)
- "Programming the Ensoniq Mirage," Keyboard Magazine, November 1986
- "EA IFF 85" Standard for Interchange Format Files, Electronic Arts, Inc. Describes the underlying conventions for all IFF files.

END OF FILE FTN.D8.0002

```
##########################################################################
### FILE: FTN.D8.0003
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $D8 (216)
Auxiliary Type: $0003

Full Name: Sampled Sound Resource
Short Name: Sound resource file

Written by : Matt Deatherage May 1992

Files of this type and auxiliary type contain sound resources.

Files of this type and auxiliary type contain one or more named sound resources. The format for sound resources (type $8024, rSoundSample) is documented in Apple IIgs Technical Note #76, Miscellaneous Resource Formats.

These files are convenient containers for files that contain sound resources, since sound resources are invisible to users who don't have resource editors or resource development tools. Since they are containers, Apple recommends that each sound resource file's name accurately reflect the contents of the file. If the file name implies a specific sound, that sound resource file should contain only that sound. If the file contains a family of sounds, such as one person pronouncing several phrases, the file name should clearly indicate the sound family is present.

Remember that users will, for the most part, only be able to move sounds by file and not by individual resources. If you have a file named "Kitten" that has two sound resources--a kitten mewing and a puppy barking--users will most likely be confused if they remove the "Kitten" file and the "Puppy" sound goes away also.

The Sound Control Panel in Apple IIgs System Software 6.0 and later (admittedly, one of the primary users of sound resources) will use sound resources in files of any type and auxiliary type, but we recommend this file type and auxiliary type for files that contain only sound resources.

Further Reference

- Apple IIgs Technical Note #76, Miscellaneous Resource Formats

```
### END OF FILE FTN.D8.0003
```

```
##########################################################################
### FILE: FTN.D8.8001
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $D8 (216)
Auxiliary Type: $8001

Full Name: HyperStudio Sampled Sound File
Short Name: HyperStudio sound

Written by: Matt Deatherage May 1990

Files of this type and auxiliary type contain sampled sounds used by the HyperStudio(TM) multimedia package.

HyperStudio is a complete hypermedia authoring system, including sound hardware, sound and graphics tools that lets you create your own hypermedia applications.

For more information on HyperStudio, contact:

Roger Wagner Publishing, Inc.
1050 Pioneer Way, Suite P
El Cajon, CA 92020
Attention: HyperStudio Technical Support
Telephone: (619) 442-0522

File Structure

HyperStudio sound files contain sampled sound data prefaced by a special HyperStudio header. The header contains information about the file and the sound, including sampling and compression information and version numbers.

The File Format

Each HyperStudio file begins with this header information.

HFileID	(+000)	4 Bytes	The ASCII characters "SSHS" ($53 $53 $48 $53). This identifies the file as a HyperStudio sound.
HDataOffset	(+004)	Long	Offset to the sound data from this field. For this version of the HyperStudio sound format, this value is always 56 ($38).
HVersID	(+008)	4 Bytes	The version of HyperStudio that created this file. The current version is represented by the four ASCII characters "RWP " ($52 $57 $50 $20).

HDataID	(+012)	4 Bytes	The ASCII characters "SSDK" ($53 $53 $44 $4B), identifying the creator of the data. Other applications wishing distinct creator IDs should contact Roger Wagner Publishing.
HLength2	(+016)	Word	Sound sample length of channel one in 256-byte pages.
HPbRate2	(+018)	Word	Playback rate for channel one. The value (HPbRate2+40) is the freqOffset for the FFStartSound toolbox call. (HPbRate2+40) * 51.40625 is the sample's frequency in Hertz.
HVolume2	(+020)	Word	Volume for channel one (0-15).
HEcho2	(+022)	Word	Echo value for channel one (0-127).
HLength	(+024)	Word	Sound sample length of channel zero in 256-byte pages. All other channels should have the same length.
HAce	(+026)	Word	Bits 0 - 13 of this word contain the ACE nBlks parameter (the number of 512-byte blocks to compress or expand). Bits 14 and 15 indicate the type of ACE compression used: %00xxxxxxxxxxxxxx - no compression %01xxxxxxxxxxxxxx - ACE method 1 (2:1) %10xxxxxxxxxxxxxx - ACE method 2 (8:3) %11xxxxxxxxxxxxxx - reserved
HPbRate	(+028)	Word	Playback rate for channel zero. The value (HPbRate+40) is the freqOffset for the FFStartSound toolbox call. (HPbRate+40) * 51.40625 is the sample's frequency in Hertz.
HVolume	(+030)	Word	Volume for channel zero (0-15).
HStereo	(+032)	Word	Number of channels for this sound file (0 = monophonic, 1 = stereo).
HEcho	(+034)	Word	Echo value for channel zero (0-127).
HReserved	(+036)	Word	Reserved for future use; set to zero.
HRepeat	(+038)	Word	Repeat count for both channels. The maximum value is 20, which indicates the sound is to repeat continuously.
HOffset1	(+040)	Long	Offset to the sampled data for channel zero. This should be the value (64+(HLength*256)).
HExtra	(+044)	Long	Reserved for future use, set to zero.
HFileName	(+048)	String	If the name of this file is 16 characters or less, you may optionally place it here as a String. If you do not place the file name here, fill these 16 bytes with zeroes.
Data	(+064)	Bytes	The start of the sampled sound data.

Further Reference

- **Apple IIgs Toolbox Reference, Volumes 2 & 3**

END OF FILE FTN.D8.8001

```
##########################################################################
### FILE: FTN.D8.xxxx
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: **$D8 (216)**
Auxiliary Type: **All**

Full Name: **Sampled Sound file**
Short Name: **Sampled Sound**

Written by: **Matt Deatherage** **January 1990**

Files of this type and auxiliary type contain sampled sound data.

Files of type $D8 should contain sampled or digitized sound data. The data is intended to be fed to sound hardware to reproduce a sound that was recorded "live." The converse to "sampled" sound is "synthesized" sound, where a computer creates wave forms and feeds them to sound hardware.

Sampled sound data can be stored in many formats. The data is traditionally sampled in discrete intervals, with a given number of bits used to record the intensity of the sound at the sampling point. In addition to the samples themselves, this requires that the file contain the sampling interval (or rate) and perhaps the number of bits used in sampling. Other information may be needed by applications, such as the duration of the sound, comment or copyright information, compression information or parameters, or "markers" which denote specific points within the sampled sound.

Apple Computer, Inc. presents a standard for such files, the Audio Interchange File Format (Audio IFF), described in another File Type Note. While Audio IFF is suitable for many needs, it cannot hope to cover all. Apple assigns auxiliary types in this file type for such purposes.

Note: Apple does not recognize a standard in which the sampling rate is contained in a file's auxiliary type. Doing so is not possible within the realm of file type and auxiliary type assignment.

The following auxiliary type assignments are current for this file type as of the publication date of this Note:

Auxiliary Type	Short Name	Developer
$0000	Audio IFF	Apple
$0002	ASIF instrument	Apple
$0003	Sampled Sound Resource	Apple
$8001	HyperStudio sound	Roger Wagner Publishing

Table 1-Auxiliary Type Assignments

The auxiliary types for this file type are reserved; any not listed in this Note or About File Type Notes must be assigned by Apple Computer, Inc. Using any file type or auxiliary type not assigned may result in conflicting identification of files by totally unrelated programs. To obtain an auxiliary type assignment in this file type, see About File Type Notes.

Further Reference

- Apple IIGS Technical Note #76, Miscellaneous Resource Formats
- File Type Note for file type $D8, auxiliary type $0000, Audio IFF File
- File Type Note for file type $D8, auxiliary type $0002, ASIF File

END OF FILE FTN.D8.xxxx

```
##########################################################################
### FILE: FTN.E0.0001
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $E0 (224)
Auxiliary Type: $0001

Full Name: AppleSingle File
Short Name: AppleSingle File

Revised by: Matt Deatherage January 1991
Written by: Matt Deatherage March 1989

Files of this type and auxiliary type contain a file in AppleSingle format.
Changes since March 1990: Added information about AppleSingle version 2.0.

AppleSingle is one of two standards (the other is AppleDouble) put forth by Apple Computer, Inc. for representing files on foreign file systems while preserving all attributes of the file's home system on file systems that do not support the same attributes.

Experience indicated that a single format would be inadequate to cover all cases. Two closely related formats, however, can serve most needs. Although the primary impetus for developing these formats is storing extended files (files with both resource and data forks) on file systems that do not support the notion of two forks, the proposed formats are general enough that theycan be used to represent a file from any file system on any other file system.

AppleSingle keeps all attributes and the contents of both forks in a single file in the foreign file system, and this Note describes this file format. AppleDouble keeps the data fork as a separate file from the file attributes and the resource fork, and is described in the File Type Notes for File Type $E0, Auxiliary Types $0002 and $0003.

AppleSingle is intended to be used primarily as a storage format, especially for cases where you must store an extended file on a foreign file system and later reconstruct the extended file. AppleDouble is more appropriate for applications where the users of the foreign file system might want to modify the contents of the file. Since most applications keep file data in the data fork, AppleDouble format saves the contents of the data fork in one file.All other file attributes, including the resource fork, are kept in a separate file.

Reasons for Using AppleSingle

There are several reasons for supporting an interchange format between file systems. Perhaps the most germane is one of the least obvious: handling extended files on foreign file systems which do not support extended files.

For example, the ProDOS FST in GS/OS can create an extended file on a ProDOS disk. However, ProDOS 8 is unable to operate on the file, since it sees itas

having an unsupported storage type. If a telecommunications program or other utility capable of transferring files is operating under ProDOS 8 andattempts to receive an extended file, it is unable to create the file.

At this point, the application could use READ_BLOCK and WRITE_BLOCK commands, along with a knowledge of the ProDOS file system, to create the file on its own. However, this is strongly discouraged. The ProDOS file system format for extended files is not documented and could change in the future. In addition, the program could be running on a eight-bit system. If the disk is only used on an eight-bit system, the extended files would not only be unwanted, but also unremovable without using the disk on an Apple IIGS or later system running GS/OS.

However, if the application is aware of the AppleSingle format, it canquickly store an extended file in AppleSingle, leaving the conversion back to the extended file to GS/OS, or another operating system. This is the recommended way for ProDOS 8 applications to create and handle extended files. Useeither AppleSingle or AppleDouble.

AppleSingle Format

An AppleSingle file contains a header followed by data. The header consists of several fixed fields and a list of entry descriptors, each pointing to an entry. Apple defines the following standard entries: Data Fork, Resource Fork, Real Name (name in the home file system), Comment, Icon and File Info. Each entry is optional, so it may not appear in the file.

Note: All numeric entries, including entries representing ProDOS data structures (such as file type and auxiliary type) are Reverse ordered. This is provided so any host CPU can attempt to interpret entries in the header without having to know the standard byte-ordering of the home file system. Therefore, in this Note you see descriptive entries like "Rev. 4 Bytes." This serves as a reminder that all header fields are stored high byte first, even though the notation Bytes does not imply any specific ordering in other File Type Notes.

Also note that ASCII strings are not stored in reverse order, just non-ASCII constants.

The Header:

Magic Number	Rev. Long	The Magic Number field is modeled after the feature in UNIX. It is intended to be used in whatever way the foreign file system distinguishes a file as AppleSingle format. See the section "Identifying AppleSingle Files." The Magic Number for AppleSingle format is $00051600, which is stored reverse as $00 $05 $16 $00 (reverse of normal 65816/6502 order).
Version Number	Rev. Long	The version of AppleSingle format, in case the format evolves (more fields may be added to the header). The version described here is $00010000, stored (reverse) as $00 $01 $00 $00.
Home File System	16 Bytes	A fixed-length, 16-byte ASCII string not

		preceded by a length byte, but possibly padded with blanks. Apple has defined these values: ProDOS $50726F444F532020202020202020202020 Macintosh $4D6163696E746F736820202020202020 MS-DOS $4D532D444F532020202020202020202020 Unix $556E9878202020202020202020202020 VAX VMS $56415820564D532020202020202020202020 Apple welcomes suggestions for other file systems that should be included in this list.
Number of entries	Rev. Word	Tells how many different entries are included in the file. This unsigned reverse word may be zero. If it is non-zero, then that number of entry descriptors immediately follows this field.

For Each Entry:

Entry ID	Rev. Long	Identifies the entry. Apple has defined the following Entry IDs and their values: 1 = Data Fork 2 = Resource Fork 3 = Real Name (The file's name in the home file system) 4 = Comment* (standard Macintosh comment) 5 = Icon, B&W* (standard Macintosh black and white icon) 6 = Icon, Color* (reserved for Macintosh color icon) 7 = File Info (file attributes,dates, etc.) 9 = Finder Info* (standard Macintosh Finder Info) Entry IDs marked with asterisks (*) are not used for most files created under ProDOS or GS/OS. Furthermore, icon entries probably do not appear in most files since they are typically stored as a bundle in the application file's resource fork on the Macintosh. Apple reserves the range of Entry IDs from $0 to $7FFFFFF for future use. The rest of the range is available for other systems to define their own entries. Apple does not arbitrate the use of the rest of the range. Descriptions of the standard entries are given below.
Offset	Rev. Long	An unsigned reverse long which indicates the byte offset from the start of the file to the start of the entry.
Entry Length	Rev. Long	An unsigned reverse long which indicates the length of the entry in bytes. The length may be zero.

Standard Entries:

The Real Name Entry:

The Real Name entry indicates the file's original filename in the host file system. This is not a Pascal or C string; it is just ASCII data. The length is indicated by the Entry Length field for the Real Name entry.

The File Info Entry:

The File Info entry (Entry ID = 7) is different for each home file system. For ProDOS files, the entry is 16 bytes long and consists of the creationdate and time and the modification date and time in ProDOS 8 (ProDOS 16/class zero GS/OS) form, the access word, a two-byte file type and four-byte auxiliary type. This is detailed in standard format below, along with defined FileInfo entries for some other file systems.

ProDOS:

Create Date	Rev. 2 Bytes	Creation date packed into standard ProDOS 8 format.
Create Time	Rev. 2 Bytes	Creation time packed into standard ProDOS 8 format.
Modification Date	Rev. 2 Bytes	Modification date packed into standard ProDOS 8 format.
Modification Time	Rev. 2 Bytes	Modification time packed into standard ProDOS 8 format.
Access	Rev. Word	The file's access. This may be used directly in ProDOS 16 or GS/OS calls; only the low byte is significant to ProDOS 8.
File Type	Rev. Word	The file type of the original file. Only the low byte is significant to ProDOS 8.
Auxiliary Type	Rev. Long	The auxiliary type of the original file. Only the low word is significant to ProDOS 8.

Note: Although the ProDOS Access field, File Type and Auxiliary Type are the same length as found in ProDOS 16 and GS/OS structures, the Create and Modification Dates and Times are stored in two-byte (albeit byte-reversed) ProDOS 8 format, not eight-byte Apple IIGS format.

Macintosh:

Create Date	Rev. Long	Unsigned number of seconds between January 1, 1904, and the creation time of this file.
Modification Date	Rev. Long	Unsigned number of seconds between January 1, 1904, and the last modification of this file.
Last Backup Date	Rev. Long	Unsigned number of seconds between January 1, 1904, and the last backup time of this file.
Attributes	Rev. Long	32 boolean flags. Once the bytes are unreversed, bit zero is the locked bit and bit one is the protected bit.

MS-DOS:

Modification Date	Rev. 4 Bytes	MS-DOS format modification date.
Attributes	Rev. 2 Bytes	MS-DOS attributes.

Unix:

Create Date/Time	Rev. 4 Bytes	Unix creation date and time.
Last Use Date/Time	Rev. 4 Bytes	Unix time for the last time this file was used.
Last Mod. Date/Time	Rev. 4 Bytes	Unix time for the last time this file was modified.

The Finder Info Entry:

The Finder Info entry (Entry ID = 9) is for files where the host file system is Macintosh. It consists of 16 bytes of Finder Info followed by 16 bytes of Extended Finder Info. These are the fields ioFlFndrInfo followed by ioFlXFndrInfo, as described in Inside Macintosh, Volume IV-183. Newlycreated files have zeroes in all Finder Info subfields. If you are creating an AppleSingle file whose home system is Macintosh, you may zero all unknown fields, but you may want to set the fdType and fdCreator subfields.

The Entries:

The entries themselves follow the header field and the entry descriptors.The actual data representing each entry must be in a single, contiguous block. The offset field in that entry's descriptor points to it. The entries could appear in any order, but since the data fork is the entry that is most commonly extended, Apple strongly recommends that the data fork always bekept last in the file to facilitate its extension. Apple also recommends that those entries that are most often read, such as Real Name, File Info (and Finder Info if present) be kept as close as possible to the header tomaximize the probability that a read of the first few blocks of the file retrieves these entries.

It is possible to have holes in the file (unused space between entries). To find the holes, you must take the list of entry descriptors and sort theminto increasing offset order. If the offset field of an entry is greater than the offset plus the length of the previous entry (sorted), then a hole exists between the entries. You can make use of such holes; for example, if afile's comment is ten bytes long, you could create a hole of 190 bytes after the comment field to easily allow for the comment to later expand to its maximum length of 200 bytes. Because an AppleSingle file may contain holes, you must find each entry by getting its offset from its entry descriptor, not by assuming that it begins after the previous entry.

Byte ordering in file header fields follows 68000 convention, and each header field has been so noted by the Reverse operator.

Identifying AppleSingle files

As this is an interchange format, from a ProDOS directory entry there is no way to guarantee which files are AppleSingle files. Apple has allocated File Type $E0, Auxiliary Type $0001 for files which are AppleSingle files. We strongly encourage ProDOS 8 and GS/OS applications to use this file type and auxiliary type assignment when creating AppleSingle files.

AppleSingle files which do not have file type $E0 and auxiliary type $0001can most easily be identified by opening them and attempting to interpret them. If they are not AppleSingle files, the Magic Number is not contained in the first four bytes of the file. The chances that the file would begin with those four bytes and not be an AppleSingle file, on a purely random basis,are 4,294,967,295 to 1. The chances that both the Magic Number and the Version bytes would be the same in a non-AppleSingle file are roughly 1.8 x 10^19 to 1.

About AppleSingle 2.0

AppleSingle 2.0 is a revision to the original AppleSingle specification described in this Note. AppleSingle 2.0 comes closer to the ideal of an interchange format by allowing file information for multiple file systems in the same AppleSingle file.

AppleSingle 2.0 basically replaces the File Info entry (ID = 7) with a File Dates entry (ID = 8) and one or more host file system entries, such as a Macintosh File Info entry (ID = 10), a ProDOS File Info entry (ID = 11), or an MS-DOS File Info entry (ID = 12). Information on these entries and AppleSingle 2.0 can be found in the AppleSingle/AppleDouble Formats for Foreign Files Developer's Note, available from APDA, AppleLink, and the Developer CD series.

Further Reference

- Inside Macintosh, Volume IV
- ProDOS 8 Technical Reference Manual
- GS/OS Reference
- AppleSingle/AppleDouble Formats for Foreign Files Developer's Note

END OF FILE FTN.E0.0001

```
##########################################################################
### FILE: FTN.E0.0002.3
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $E0 (224)
Auxiliary Types: $0002 & $0003

Full Name: AppleDouble Header File (Auxiliary Type $0002)
AppleDouble Data File (Auxiliary Type $0003)
Short Name: AppleDouble Header (Auxiliary Type $0002)
AppleDouble Data (Auxiliary Type $0003)

Revised by: Matt Deatherage November 1990
Written by: Matt Deatherage March 1989

Files of these types and auxiliary types contain file data in AppleDouble format.
Changes since March 1990: Added information about AppleDouble 2.0.

AppleDouble is one of two standards (the other is AppleSingle) put forth by Apple Computer, Inc. for representing files on foreign file systems while preserving all attributes of the file's home system on file systems that do not support the same attributes.

Experience indicated that a single format would be inadequate to cover all cases. Two closely related formats, however, can serve most needs. Although the primary impetus for developing these formats is storing extended files (files with both resource and data forks) on file systems that do not support the notion of two forks, the proposed formats are general enough that theycan be used to represent a file from any file system on any other file system.

AppleDouble keeps the data fork as a separate file from the file attributes and the resource fork, and this Note describes this file format. AppleSingle keeps all attributes and the contents of both forks in a single file in the foreign file system, and is described in the File Type Note for File Type$E0, Auxiliary Type $0001.

AppleSingle is intended to be used primarily as a storage format, especially for cases where you must store an extended file on a foreign file system and later reconstruct the extended file. AppleDouble is more appropriate for applications where the users of the foreign file system might want to modify the contents of the file. Since most applications keep file data in the data fork, AppleDouble format saves the contents of the data fork in one file.All other file attributes, including the resource fork, are kept in a separate file.

Reasons for Using AppleDouble

There are several reasons for supporting an interchange format between file systems. Perhaps the most germane is one of the least obvious: handling extended files on foreign file systems which do not support extended files.

For example, the ProDOS FST in GS/OS can create an extended file on a ProDOS disk. However, ProDOS 8 is unable to operate on the file, since it sees itas having an unsupported storage type. If a telecommunications program or other utility capable of transferring files is operating under ProDOS 8 andattempts to receive an extended file, it is unable to create the file.

At this point, the application could use READ_BLOCK and WRITE_BLOCK commands, along with a knowledge of the ProDOS file system, to create the file on its own. However, this is strongly discouraged. The ProDOS file system format for extended files is not documented and could change in the future. In addition, the program could be running on a eight-bit system. If the disk is only used on an eight-bit system, the extended files would not only be unwanted, but also unremovable without using the disk on an Apple IIGS or later system running GS/OS.

However, if the application is aware of the AppleDouble format, it canquickly store an extended file in AppleDouble, leaving the conversion back to the extended file to GS/OS, or another operating system. This is the recommended way for ProDOS 8 applications to create and handle extended files. Useeither AppleSingle or AppleDouble.

AppleDouble Format

AppleDouble consists of two files, an AppleDouble Header File and an AppleDouble Data File. The AppleDouble Header file contains a headerfollowed by data. The header consists of several fixed fields and a list of entry descriptors, each pointing to an entry. Apple defines these standardentries: Resource Fork, Real Name (name in the home file system), Comment, Icon and File Info. Each entry is optional, so it may not appear in the file. Wealso define the new entry Data Pathname, pointing to the pathname of the AppleDouble Data File. The Header File has exactly the same format as an AppleSingle file, except it has no data fork entry. The AppleDouble DataFile consists of just the data fork of the file, with no extra header at all.

Note: All numeric entries, including entries representing ProDOS data structures (such as file type and auxiliary type) are Reverse ordered. This is provided so any host CPU can attempt to interpret entries in the header without having to know the standard byte-ordering of the home file system. Therefore, in this Note you see descriptive entries like "Rev. 4 Bytes." This serves as a reminder that all header fields are stored high byte first, even though the notation Bytes does not imply any specific ordering in other File Type Notes.

Also note that ASCII strings are not stored in reverse order, just non-ASCII constants.

The Header in the Header File:

Magic Number	Rev. Long	The Magic Number field is modeled after the feature in UNIX. It is intended to be used in whatever way the foreign file system distinguishes a file as AppleDouble format. See the section "Identifying AppleDouble Files." The Magic Number for AppleDouble format is $00051607, which is

		stored reverse as $00 $05 $16 $07 (reverse of normal 65816/6502 order).
Version Number	Rev. Long	The version of AppleDouble format, in case the format evolves (more fields may be added to the header). The version described here is $00010000, stored (reverse) as $00 $01 $00 $00.
Home File System	16 Bytes	A fixed-length, 16-byte ASCII string not preceded by a length byte, but possibly padded with blanks. Apple has defined these values: ProDOS $50726F444F532020202020202020202020 Macintosh $4D6163696E746F736820202020202020 MS-DOS $4D532D444F532020202020202020202020 Unix $556E98782020202020202020202020202020 VAX VMS $56415820564D532020202020202020202020 Apple welcomes suggestions for other file systems that should be included in this list.
Number of entries	Rev. Word	Tells how many different entries are included in the file. This unsigned reverse word may be zero. If it is non-zero, then that number of entry descriptors immediately follows this field.

For Each Entry:

Entry ID	Rev. Long	Identifies the entry. Apple has defined the following Entry IDs and their values: 1 = Data Fork 2 = Resource Fork 3 = Real Name (The file's name in the home file system) 4 = Comment* (standard Macintosh comment) 5 = Icon, B&W* (standard Macintosh black and white icon) 6 = Icon, Color* (reserved for Macintosh color icon) 7 = File Info (file attributes,dates, etc.) 9 = Finder Info* (standard Macintosh Finder Info) Entry IDs marked with asterisks (*) are not used for most files created under ProDOS or GS/OS. Furthermore, icon entries probably do not appear in most files since they are typically stored as a bundle in the application file's resource fork on the Macintosh. Apple reserves the range of Entry IDs from $0 to $7FFFFFFF for future use. The rest of the range is available for other systems to define their own entries. Apple does not arbitrate the use of the rest of the range. Descriptions of the standard entries are

		given below.
Offset	Rev. Long	An unsigned reverse long which indicates the byte offset from the start of the file to the start of the entry.
Entry Length	Rev. Long	An unsigned reverse long which indicates the length of the entry in bytes. The length may be zero.

Standard Entries:

The Real Name Entry:

The Real Name entry indicates the file's original filename in the host file system. This is not a Pascal or C string; it is just ASCII data. The length is indicated by the Entry Length field for the Real Name entry.

The File Info Entry:

The File Info entry (Entry ID = 7) is different for each home file system. For ProDOS files, the entry is 16 bytes long and consists of the creationdate and time and the modification date and time in ProDOS 8 (ProDOS 16/class zero GS/OS) form, the access word, a two-byte file type and four-byte auxiliary type. This is detailed in standard format below, along with defined FileInfo entries for some other file systems.

ProDOS:

Create Date	Rev. 2 Bytes	Creation date packed into standard ProDOS 8 format.
Create Time	Rev. 2 Bytes	Creation time packed into standard ProDOS 8 format.
Modification Date	Rev. 2 Bytes	Modification date packed into standard ProDOS 8 format.
Modification Time	Rev. 2 Bytes	Modification time packed into standard ProDOS 8 format.
Access	Rev. Word	The file's access. This may be used directly in ProDOS 16 or GS/OS calls; only the low byte is significant to ProDOS 8.
File Type	Rev. Word	The file type of the original file. Only the low byte is significant to ProDOS 8.
Auxiliary Type	Rev. Long	The auxiliary type of the original file. Only the low word is significant to ProDOS 8.

Note: Although the ProDOS Access field, File Type and Auxiliary Type are the same length as found in ProDOS 16 and GS/OS structures, the Create and Modification Dates and Times are stored in two-byte (albeit byte-reversed) ProDOS 8 format, not eight-byte Apple IIGS format.

Macintosh:

Create Date	Rev. Long	Unsigned number of seconds between January 1, 1904, and the creation time of this file.
Modification Date	Rev. Long	Unsigned number of seconds between January 1, 1904, and the last modification

		of this file.
Last Backup Date	Rev. Long	Unsigned number of seconds between January 1, 1904, and the last backup time of this file.
Attributes	Rev. Long	32 boolean flags. Once the bytes are unreversed, bit zero is the locked bit and bit one is the protected bit.

MS-DOS:

Modification Date	Rev. 4 Bytes	MS-DOS format modification date.
Attributes	Rev. 2 Bytes	MS-DOS attributes.

Unix:

Create Date/Time	Rev. 4 Bytes	Unix creation date and time.
Last Use Date/Time	Rev. 4 Bytes	Unix time for the last time this file was used.
Last Mod. Date/Time	Rev. 4 Bytes	Unix time for the last time this file was modified.

The Finder Info Entry:

The Finder Info entry (Entry ID = 9) is for files where the host file system is Macintosh. It consists of 16 bytes of Finder Info followed by 16 bytes of Extended Finder Info. These are the fields ioFlFndrInfo followed by ioFlXFndrInfo, as described in Inside Macintosh, Volume IV-183. Newlycreated files have zeroes in all Finder Info subfields. If you are creating an AppleDouble file whose home system is Macintosh, you may zero all unknown fields, but you may want to set the fdType and fdCreator subfields.

The Data Pathname Entry:

The Data Pathname entry (Entry ID = 100) is defined for the first time inthis Note. It consists of a class one GS/OS input string noting the pathname of the AppleDouble Data File as originally created:

Path Length	Rev. Word	The length of the pathname.
Pathname	Bytes	ASCII pathname of the AppleDouble Data File when created.

For strategies on using this segment (or not using it) to find theAppleDouble Data File, see the section "Finding the AppleDouble Data File."

The Entries in the Header File:

The entries themselves follow the header field and the entry descriptors.The actual data representing each entry must be in a single, contiguous block. The offset field in that entry's descriptor points to it. The entries could appear in any order, but since the data fork is the entry that is most commonly extended, Apple strongly recommends that the data fork always bekept last in the file to facilitate its extension. Apple also recommends that those entries that are most often read, such as Real Name, File Info (and Finder Info if present) be kept as close as possible to the header tomaximize the probability that a read of the first few blocks of the file retrieves these entries.

It is possible to have holes in the file (unused space between entries). To find the holes, you must take the list of entry descriptors and sort theminto increasing offset order. If the offset field of an entry is greater than the offset plus the length of the previous entry (sorted), then a hole exists between the entries. You can make use of such holes; for example, if afile's comment is ten bytes long, you could create a hole of 190 bytes after the comment field to easily allow for the comment to later expand to its maximum length of 200 bytes. Because an AppleDouble file may contain holes, you must find each entry by getting its offset from its entry descriptor, not by assuming that it begins after the previous entry.

Byte ordering in file header fields follows 68000 convention, and each header field has been so noted by the Reverse operator.

The AppleDouble Data File

The AppleDouble Data File is simply the data fork of the original file contained in a file of its own. You may create it with a File Type and Auxiliary Type assignment best suited to it, if desired. For example, if the program creating the AppleDouble Data File knows that the data fork contains strictly ASCII text, it could create the file with File Type $04 (Text File) so that other applications can deal with it accordingly.

If the creating program wishes to make no assumptions about the content ofthe data fork, it is encouraged to create the AppleDouble Data File with filetype $E0 and auxiliary type $0003. This identifies the file as an AppleDoubleData File.

Identifying AppleDouble Files

As this is an interchange format, from a ProDOS directory entry there is no way to guarantee which files are AppleDouble files. Apple has allocated File Type $E0, Auxiliary Type $0002 for files which are AppleDouble Header Files, and File Type $E0, Auxiliary Type $0003 for files which are AppleDouble Data Files. We strongly encourage ProDOS 8 and GS/OS applications to use these file type and auxiliary type assignments when creating AppleDouble files.

AppleDouble files which do not have file type $E0 and auxiliary type $0002 or $0003 can most easily be identified by opening them and attempting to interpret them. If it is not an AppleDouble Header File, the Magic Number is not contained in the first four bytes of the file. The chances that the file would begin with those four bytes and not be an AppleDouble Header File, on a purely random basis, are 4,294,967,295 to 1. The chances that both the Magic Number and the Version bytes would be the same in a non-AppleSingle file are roughly 1.8 x 10^19 to 1.

Finding the AppleDouble Data File

Since the AppleDouble Data File can be stored anywhere, with any file typeand auxiliary type, a program may have to make an effort to find it. Werecommend the following steps:

1. If the Data Pathname segment exists, use that pathname. If the path specified in the segment does not exist, extract the file

name from the end of the pathname and look in the current directory for that file name.
2. If Step 1 does not find the file (or if the Data Pathname segment does not exist), perform the appropriate Home File System algorithm (described below) to generate the name of the AppleDouble Data File from the AppleDouble Header File.
3. If none of the file names generated in Step 2 are found, ask the user where the AppleDouble Data File is located.

Filename Conventions:

Apple proposes the following standard for identifying AppleDouble Header File names and AppleDouble Data File names from the file's real name.

ProDOS:

To generate the AppleDouble Data File name, use character substitution or deletion to remove illegal characters, and use truncation if necessary to reduce the length of the name to two characters less than the maximum file name length. This would be a maximum of 13, since the maximum file name length is 15.

To generate the AppleDouble Header File name, prefix the AppleDouble DataFile name with the characters "R." (uppercase R period).

For example, the file name "This is a Foo File" could translate to an AppleDouble Data File Name of "THIS.IS.A.FOO." The AppleDouble Header File name would then be "R.THIS.IS.A.FOO."

Unix:

To generate the AppleDouble Data File name, use character substitution to replace any illegal characters with an underscore (_). Since different Unix systems have different requirements on maximum file name length, do not explicitly truncate the name to a specific length. Rather, allow the truncation to be done by the Unix functions create(), open(), etc.

To generate the AppleDouble Header File name, A/UX (Apple's implementation of Unix for Macintosh computers) prefixes a percent sign (%) to the AppleDouble Data File name. If necessary, truncate the last character to keep the file name within the legal length range. Other Unix systems may prefix adirectory name (e.g., ".AppleDouble/") to the AppleDouble Data File name to create the name of the AppleDouble Header File. In this scheme, all AppleDouble Header Files corresponding to AppleDouble Data files are kept together in a single subdirectory.

MS-DOS:

To generate the AppleDouble Data File name, use character substitution or deletion to remove illegal characters, and use truncation if necessary to reduce the length of the name to eight characters. Then add the MS-DOS extension that is most appropriate to the file (such as "TXT" for a pure text file).

To generate the AppleDouble Header File name, add the extension ".ADF" to the eight-character file name.

In any instance, most programs probably wish to display the names being used for both AppleDouble files, so that the user may keep track of them on disk.

AppleDouble name derivations will be defined for all other file systems of interest. This allows applications running on the foreign file system (and users as well) to see easily which files are AppleDouble pairs.Knowledgeable users, if they know the derivation, could rename or move the files so as to preserve the connection between the two. However, there is no guaranteed way to prevent one file of the pair from being inconsistently renamed, moved, or deleted.

About AppleDouble 2.0

AppleDouble 2.0 is a revision to the original AppleDouble specification described in this Note. AppleDouble 2.0 comes closer to the ideal of an interchange format by allowing file information for multiple file systems in the same AppleDouble file.

AppleDouble 2.0 basically replaces the File Info entry (ID = 7) with a File Dates entry (ID = 8) and one or more host file system entries, such as a Macintosh File Info entry (ID = 10), a ProDOS File Info entry (ID = 11), or an MS-DOS File Info entry (ID = 12). Information on these entries and AppleDouble 2.0 can be found in the AppleSingle/AppleDouble Formats for Foreign Files Developer's Note, available from APDA, AppleLink, and the Developer CD series.

Further Reference

- Inside Macintosh, Volume IV
- ProDOS 8 Technical Reference Manual
- GS/OS Reference
- AppleSingle/AppleDouble Formats for Foreign Files Developer's Note

END OF FILE FTN.E0.0002.3

```
##########################################################################
### FILE: FTN.E0.0005
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: **$E0 (224)**
Auxiliary Type: **$0005**

Full Name: **DiskCopy disk image**
Short Name: **DiskCopy disk image**

Written by: **Matt Deatherage, Dave Lyons & Steve Christensen** **May 1992**

Files of this type and auxiliary type contain disk images from Apple's DiskCopy program on the Macintosh.

DiskCopy is a program written by Steve Christensen of Apple Computer, Inc., for internal use in duplicating and distributing 3.5" floppy disks. Because of its utility in distributing disk images on the Macintosh, DiskCopy is used in several Apple developer products even though DiskCopy is not an official Apple product and is not supported as such.

Since the monthly Developer CD Series discs contain many DiskCopy disk images, and since the AppleShare and HFS FSTs in System Software 6.0 and later automatically translate DiskCopy files (HFS file type dImg and creator dCpy) to Apple II file type $E0 and auxiliary type $0005, the format is provided here for your utility use only. Apple does not guarantee that files not generated by DiskCopy will work with DiskCopy.

DEFINITIONS

DiskCopy uses a simple checksum algorithm to help insure data integrity for archived disk images. The algorithm for generating the 32-bit checksum is as follows:

```
Initialize checksum to zero
For each data REVERSE WORD:
     Add the data REVERSE WORD to the checksum
     Rotate the 32-bit checksum right one bit (wrapping bit 0 to bit 31)
```

The following 65816 assembly language routine calculates a DiskCopy checksum. It's not a speedy operation--it takes about 12 seconds to calculate the checksum on an 800K disk image. Anyone finding an assembly routine that can perform this task in under 5 seconds may apply for their IIgs Certificate of Deityship, as documented in the File Type Note for file type $B6.

(Oh, by the way, any entries have to be under 1K in size--the following routine is 88 bytes. So don't think unwinding loops is your ticket to fame and fortune.)

****

```
*
* Compute checksum for DiskCopy data
*
* v1.2  by David A. Lyons, 18-May-92
*
* MPW IIgs assembly format
*
* Inputs on stack:
*    Push pointer to data (long)
*    Push size of data (long)  (Must be even!)
*    JSL CalcChecksum
*    STA TheChecksum+2
*    STX TheChecksum
*
* Output:
*    Checksum in A and X (bytes +0 and +1 in X, bytes +2 and +3 in A)
*    (The inputs have been removed from the stack)
*
********************************************************************************
CalcChecksum        PROC
                    phd                 ;save caller's direct page reg
                    lda #0
                    pha
                    pha                 ;push initial checksum value (zero)
                    tsc
                    tcd

checksum            equ 1
oldD                equ checksum+4
theRTL              equ oldD+2
dataSize            equ theRTL+3
dataPtr             equ dataSize+4

*** Set dataSize to -(dataSize/2)-1 so we can count up by one
*** (instead of down by two) to see when we're done
                    lda <dataSize+2
                    lsr a
                    eor #$ffff
                    sta <dataSize+2
                    lda <dataSize
                    ror a
                    eor #$ffff
                    sta <dataSize

                    ldy #0
nextWord            inc <dataSize
                    bne moreData
                    inc <dataSize+2
                    beq noMoreData
moreData

*** Get next 16-bit word from the data buffer
                    lda [<dataPtr],y
                    xba                 ;swap to 65816 byte order

*** Add the data word to the checksum
                    clc
                    adc <checksum
```

```
                    sta <checksum
                    bcc noCksumRoll
                    inc <checksum+2
noCksumRoll
*** Rotate the 32-bit checksum right one bit, wrapping bit 0 into bit 31
                    lda <checksum+2
                    lsr a
                    ror <checksum
                    bcc bit0was0
                    ora #$8000          ;if we rotated a 1 out of bit 0,
bit0was0            sta <checksum+2     ;  then set bit 31

*** Advance to the next word and go back for more
                    iny
                    iny
                    bne nextWord        ;go back for more data
                    inc <dataPtr+2
                    bra nextWord        ;go back for next bank of data

noMoreData          pla
                    xba
                    tay
                    pla
                    xba
                    tax                 ;pull checksum into YX (put in 68000
order)

                    pld                 ;restore caller's direct page reg

                    lda 2,s
                    sta 2+8,s
                    lda 1,s
                    sta 1+8,s
                    pla
                    pla
                    pla
                    pla                 ;discard input values

                    tya
                    rtl

                    EndP

                    END
```

The following definition is used in this document in addition to those defined for all Apple II file types:

Checksum A 32-byte quantity calculated using the previously-defined algorithm. When these are contained in the file, they are in REVERSE order.

FILE STRUCTURE

All of the information for a DiskCopy disk image is in the data fork. The resource fork usually contains Macintosh resources (in Macintosh resource fork format), including vers resources listing the checksums. This allows

Macintosh users to use the Macintosh Finder's "Get Info..." function to quickly examine the checksums.

The File Format

Because this is a native Macintosh file format, all the multi-byte constants are stored in Reverse order.

diskName	(+000)	64 Bytes	A Pascal String containing the name of the disk. This field takes 64 bytes regardless of the length of the String.
dataSize	(+064)	Rev. Long	The number of bytes (not blocks) of user data. User data is the 512 bytes of each block that a normal block-reading command returns.
tagSize	(+068)	Rev. Long	The number of bytes of tag data. Tag data is the extra 12 bytes of "scavenger" information present on 400K and 800K Macintosh disks. Apple II operating systems always leave these bytes zeroed, and they're not present on 720K or 1440K disks. If there are no tag bytes, this field will be zero.
dataChecksum	(+072)	Checksum	Checksum of all the user data on the disk. The checksum algorithm is called for the entire disk, not on a block-by-block or sector-by-sector basis. This is in Reverse order (most significant byte first).
tagChecksum	(+076)	Checksum	Checksum of all the tag data on the disk. If there is no tag data, this should be zero. This is in Reverse order (most significant byte first).
diskFormat	(+080)	Byte	0 = 400K 1 = 800K 2 = 720K 3 = 1440K (all other values are reserved)
formatByte	(+081)	Byte	$12 = 400K $22 = >400K Macintosh (DiskCopy uses this value for all Apple II disks not 800K in size, and even for some of those) $24 = 800K Apple II disk
private	(+082)	Rev. Word	Must be $0100. If this field is not $0100, the file may be in a different format.
userData	(+084)		dataSize Bytes The data blocks for the disk. These are in order from block zero through the end of the disk.
tagData	(+xxx)	tagSize Bytes	The tag data for this disk, starting with the tag data for the first block and proceeding in order. This field is not present for 720K and 1440K disks, but it is present for all other formats even if all the data is zeroes.

Further Reference

- GS/OS Reference

END OF FILE FTN.E0.0005

```
##########################################################################
### FILE: FTN.E0.8000
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: **$E0 (224)**
Auxiliary Type: **$8000**

Full Name: **Binary II File**
Short Name: **Binary II File**

Written by: **Matt Deatherage** **July 1989**

Files of this type and auxiliary type contain other files with their attributes encoded in Binary II format.

Binary II is a widely used and accepted standard for keeping file attributes with files as they are transferred, usually by modem or other form of telecommunication. Files that are known Binary II files should be written to disk with file type $E0 and auxiliary type $8000 as a clear indication to other programs that the file contains files with Binary II specifications.

Binary II was developed by Gary B. Little, author of the Point-To-Point communication's product and author of several Apple II reference books. He is also Apple's Product Manager for third-party Development Tools and Languages. Gary welcomes your comments and suggestions on the Binary II standard at the following address:

Gary B. Little
3304 Plateau Drive
Belmont, CA 94002

AppleLink:	**LITTLE**
AppleLink--Personal Edition:	**GaryLittle**
CompuServe:	**70135,1007**
GEnie:	**GARY.LITTLE**

Why Binary II?

Transferring Apple II files in binary form to commercial information services and bulletin boards (referred to in this Note as "hosts") can be, to put it mildly, a frustrating exercise. Although most hosts are able to receive a file's data in binary form (using protocols such as XMODEM), they don't receive the file's all-important attribute bytes. All the common Apple II file system, notably the ProDOS file system, store the attributes inside the disk directory, not inside the file itself.

The ProDOS attributes are the access code, file type code, auxiliary type code, storage type code, date of creation and last modification, time of creation and last modification, the file size, and the name of the file

itself. Under GS/OS, the same parameters exist for other file systems as well as file system-specific information and two-forked file information. It is usually not possible to use a ProDOS file's data without knowing the file's attributes (particularly the file type, auxiliary type, and size). Therefore, ProDOS files uploaded in binary format to a host are useless to those who download them. The same is true for DOS 3.3 and Pascal files.

Many Apple II communication programs use special protocols for transferring file attributes during a binary file transfer, but none of these protocols have been implemented by hosts. These programs are only useful for exchanging files with another Apple II running the same program.

Without a standard like Binary II, the only acceptable way to transfer an Apple II file to a host is to convert it into ASCII text before sending it. Such a text file would contain a listing of an AppleSoft program, or a series of Apple II monitor commands (e.g., 300:A4 32). Someone downloading a file can convert it to binary form using the AppleSoft EXEC command.

The main disadvantage of this technique is that the text version of the file is over three times the size of the original binary file, making it expensive (in terms of time and money) to upload and download. It is also awkward, and sometimes impossible, to perform the binary-to-text or text-to-binary conversion.

The solution to the problem is to upload an encoded binary file which contains not just the file's data, but the file's attributes as well. Someone downloading such a file can then use a conversion program to strip the attributes from the file and create a file with the required attributes.

This Note describes such a format: Binary II. The description of the format is detailed for the purpose of allowing software developers to implement it in Apple II communication programs.

What Binary II is Not

Binary II is not an archival or compression standard. It is designed to be a simple method to keep the attributes normally in a disk file's directory entry with the file as it is transferred. Although multiple files may be placed together with Binary II, this is a matter of convenience for telecommunication programs.

A true archival standard must be designed as such, with the capability to manipulate files within the archive as well as linking them together (compressed or uncompressed) for transfer. NuFX (documented in Apple II File Type Note for File Type $E0, Auxiliary Type $8002) is a good example of a robust, full-featured Apple II archival standard.

Binary II is primarily designed to be added to and subtracted from files "on-the-fly" by telecommunication programs. Binary II files should only be found on disks when they are transferred by a telecommunication program that does not have Binary II capabilities, in which case a separate utility (such as Binary Library Utility by Floyd Zink, Jr.) must be used to extract the files. Telecommunication programs should be able to transfer files without Binary II processing, however, they should support Binary II processing as a default.

The Binary II File Format

The Binary II form of a standard file consists of a 128-byte file information header followed by the file's data. The data portion of the file is padded with nulls ($00 bytes), if necessary, to ensure the data length is an even multiple of 128 bytes.

The file information header contains four ID bytes, the attributes of the file (in ProDOS 8 form), and some control information.

The structure of the header is as follows:

+000	ID Bytes	3 Bytes	These three bytes are always $0A $47 $4C for identification purposes, so programs may recognize Binary II files as they are received.
+003	Access Code	Byte	ProDOS 8 access byte.
+004	File Type	Byte	ProDOS 8 file type.
+005	Aux Type	Word	ProDOS 8 auxiliary type.
+007	Storage Type	Byte	ProDOS 8 storage type value.
+008	File Size	Word	The size of the file in 512-byte blocks.
+010	Mod. Date	2 Bytes	Date of modification, in ProDOS 8 compressed format.
+012	Mod. Time	2 Bytes	Time of modification, in ProDOS 8 compressed format.
+014	Create Date	2 Bytes	Date of creation, in ProDOS 8 compressed format.
+016	Create Time	2 Bytes	Time of creation, in ProDOS 8 compressed format.
+018	ID Byte	Byte	A fourth ID byte. This must always be $02.
+019	Reserved	Byte	Reserved, must be set to zero.
+020	EOF	3 Bytes	The end-of-file value for the file (low byte first).
+023	File Name	String	Pascal string containing the ASCII filename or partial pathname of this file in ProDOS 8 format. The string cannot be longer than 64 characters.

If the File Name String is a filename and not a partial pathname, then the following optional parameter may be supplied:

+039	Native Name String		Pascal string containing the ASCII value of the native filename. This string may not be longer than 48 characters, and will not be present if the length byte of File Name (+023) is larger than 15 ($0F). If this field is specified, the File Name field must contain a filename, not a partial pathname.
+088	Reserved	21 Bytes	Reserved. These bytes must be set to zero for future compatibility.
+109	GAux Type	Word	The high word of the file's GS/OS auxiliary type.
+111	GAccess	Byte	The high byte of the file's GS/OS access word.

+112	GFile Type	Byte	The high byte of the file's GS/OS file type.
+113	GStorage	Byte	The high byte of the file's GS/OS storage type.
+114	GFile Size	Word	The high word of the GS/OS file's size in 512-byte blocks.
+116	GEOF	Byte	The high byte of the file's GS/OS EOF value.
+117	Disk Space	Long	The number of 512-byte disk blocks the files inside the Binary II file will occupy after they've been removed from the Binary II file. (The format of a Binary II file containing multiple files is described later in this Note.) If the number is zero, the creator of the Binary II file didn't bother to calculate the space needed. This value must be placed in the file information header for the first file inside the Binary II file; it can be set to zero in subsequent headers. A downloading program can inspect Disk Space and abort the transfer immediately if there isn't enough free space on the disk.
+121	OS Type	Byte	This value indicates the native operating system of the file: $00 ProDOS or SOS $01 DOS 3.3 $02 Reserved $03 DOS 3.2 or DOS 3.1 $04 Apple II Pascal $05 Macintosh MFS $06 Macintosh HFS $07 Lisa Filing System $08 Apple CP/M $09 Reserved (returned by the GS/OS Character FST) $0A MS-DOS $0B High Sierra (CD-ROM) $0C ISO 9660 (CD-ROM) $0D AppleShare Note this list is slightly different (in the first three entries) from the standard GS/OS file system ID list. A GS/OS communication program should not place a zero in this field unless the file's native file system truly is ProDOS. The file's native file system is returned in the file_sys_id parameter from the GetDirEntry call.
+122	Native File Type	Word	This has meaning only if OS Type is non-zero. If so, it is set to the actual file type code assigned to the file by it's native operating system. (Some operating systems, such as MS-DOS and CP/M, do not use file type codes, however.) Contrast this with the File Type at +004, which is

		the closest equivalent ProDOS file type. The Native File Type is needed to distinguish files which have the same ProDOS file type, but which may have different file types in their native operating system. Note that if the file type code is only one byte long (the usual case), the high-order byte of Native File Type is set to zero.
+124	Phantom File Flag Byte	This byte indicates whether a receiver of the Binary II file should save the file which follows (flag is zero) or ignore it (flag is non-zero). It is anticipated that some communication programs will use phantom files to pass non-essential explanatory notes or encoded information which would be understood only by a receiver using the same communication program. Such programs must not rely on receiving a phantom file, however, since this would mean they couldn't handle Binary II files created by other communication programs. Phantom Files may also be used to pass extended file attributes when available. The first two bytes in a phantom file must contain an ID code unique to the communication program, or a universal identifier concerning the contents of the phantom file. Developers must obtain ID codes from Gary Little to ensure uniqueness (see the beginning of this Note for his address). Here is a current list of approved ID codes for phantom files used by Apple II communication programs: $00 $00 ASCII text terminated with a zero byte. $00 $01 Point-to-Point $00 $02 Tele-Master Communications System $00 $03 ProTERM $00 $04 Modem MGR $00 $05 CommWorks $00 $06 MouseTalk $01 $00 Option_list data (see later in this Note). The ID bytes are the first two bytes of the phantom file.
+125	Data Flags Flag Byte	Bit 7: 1 = file is compressed Bit 6: 1 = file is encrypted Bits 5-1: Reserved Bit 0: 1 = file is sparse A Binary II downloading program can examine this byte and warn the user that the file must be expanded, decrypted or unpacked. The person uploading a Binary

			II file may use any convenient method for compressing, encrypting, or packing the file but is responsible for providing instructions on how to restore the file to its original state.
+126	Version	Byte	This release of Binary II has a version number of $01.
+127	Number of Files to Follow	Byte	An appealing feature of Binary II is that a single Binary II file can hold multiple disk files, making it easy to keep a group of related files "glued" together when they're sent to a host. This byte contains the number of files in this Binary II file that are behind it. If this is the first file in a Binary II file containing three disk files, this byte would be $02. The second disk file in the same Binary II file would have a value of $01 in this parameter, and the last would have value $00. This count tells the Binary II downloading program how many files are remaining. If any phantom files are included, they must be included in this count.

Filenames and Partial Pathnames

You can put a standard ProDOS filename or a partial pathname in the file information header (but never a complete pathname). Don't use a partial pathname unless you've included, earlier in the Binary II file, file information headers for each of the directories referred to in the partial pathname. Such a header must have its "end of file position" bytes set to zero, and no data blocks for the subdirectory file must follow it.

For example, if you want to send a file whose partial pathname is HELP/GS/READ.ME, first send a file information header defining the HELP/ subdirectory, then one defining the HELP/GS/ subdirectory. If you don't, someone downloading the Binary II file won't be able to convert it because the necessary subdirectories will not exist.

Note: GS/OS communication programs must use the slash (/) as the pathname's separator in any partial pathname it puts in the header. Since GS/OS's standard separator is the colon (:), a conversion may be necessary.

Filename Convention

Whenever a file is sent to a host, the host asks the sender to provide a name for it. If it's a file in Binary II form, the name provided should end in .BNY so its special form will be apparent to anyone viewing a list of filenames. If the file is compacted (using the public-domain Squeeze algorithm) before being converted to Binary II form, use a .BQY suffix instead. If the file is a NuFX archive, use the suffix .BXY.

Identifying Binary II Files

You can determine, while transferring, if a file is in Binary II form by examining the ID bytes at offsets +000, +001, +002 and +018 from the beginning of the file. They must be $0A, $47, $4C and $02, respectively.

Once Binary II files are identified, you can use the data in the file information header to create and open a ProDOS file with the correct name and attributes, transfer the file data in the Binary II file to the ProDOS file, set the ProDOS file size, then close the ProDOS file. You would repeat this for each file contained inside the Binary II file.

Note: The number of 128-byte blocks following the file information header must be derived from the EOF attribute for the file. Calculate the number by dividing the EOF by 128 and adding one to the result if EOF is not 0 or an exact multiple of 128. However, if the file information header defines a subdirectory (the file type is $0F), simple create the subdirectory file. Do not open it and do not try to set its size.

Ideally, all this conversion work will be done automatically by a communication program during file transfer. If not, a separate conversion program (such as the previously mentioned Binary Library Utility, or BLU) must be used to do this for you.

Option_List Phantom Files

GS/OS will return, when asked, an option_list for files on many file calls. The option_list consists of a Word buffer length (which must be at least $2E), followed by a Word number of bytes GS/OS put in the buffer, a Word GS/OS file system identification, and the given number of bytes of FST-specific information (minus two; the count GS/OS returns includes the file system identifier).

Option_list values are FST specific and contain values important to the native file system but not important to GS/OS. For AppleShare, the option_list contains Finder Information, parent directory identification, and access privileges. This information should be transferred with files.

Binary II uses a phantom file with identifier $01 $00 to indicate an option_list. When this phantom file is seen, the contents should be used as the option_list for the file that immediately follows this file in the Binary II file. The other attributes of the phantom file must be set to the same values as those for the file immediately following (the file for which the phantom file contains the option_list). The EOF for the phantom file must be the size of the option_list + 2, and the file size must be adjusted accordingly to account for the phantom file ID bytes.

When receiving a Binary II file, the contents of this phantom file should be used as option_list input on a GS/OS SetFileInfo call.

If the GS/OS option_list returns a total of two bytes (just the file_sys_ID), there is no FST-specific information, and the option_list phantom file may safely be omitted.

The format of the option_list phantom file is as follows:

+000	Phantom ID	2 Bytes	The identifying bytes $01 $00.
+002	List Size	Word	The length of the bytes in the option_list, starting with the file system ID (the next word).
+004	FileSysID	Word	A GS/OS (not Binary II) file_sys_ID for the volume on which the file was stored.
+006	List Bytes	Bytes	The bytes of the option list. There should be (List Size) of them, counting the previous word (FileSysID).

Extended File Considerations

Extended files contain two logical segments: a data fork and a resource fork. These files can be created and manipulated by GS/OS, but not by ProDOS 8 or any other Apple II operating system.

When a GS/OS-based communication program sends an extended file, it must send it in the AppleSingle file format, preceded by a Binary II file information header. (Such a program could easily convert an extended file to AppleSingle format on the fly.) The Binary II header must contain the attributes of the AppleSingle file (including a file type of $E0 and an auxiliary type of $0001) and the "storage type code" field must be $01. (The EOF positions for the data fork and resource fork of the extended file appear in an entry in the AppleSingle file header, not in the Binary II header.)

The AppleSingle format is described in Apple II File Type Note for File Type $E0, Auxiliary Type $0001.

A GS/OS-based communication program that receives an AppleSingle file can easily convert it on the fly to the extended file it defines. ProDOS 8-based communication programs can only save the AppleSingle file to disk because it's not possible (nor is it encouraged to attempt) to create extended files with ProDOS 8 (without using block-level calls); a GS/OS based utility program is needed to convert the AppleSingle file to its extended form.

DOS 3.3 Considerations

With a little extra effort, you can also convert DOS 3.3 files to Binary II form. This involves translating the DOS 3.3 file attributes to the corresponding ProDOS attributes so that you can build a proper file information header.

- Set the name to one that adheres to the stricter ProDOS naming rules and put its length at +023 and the name itself at +024 to +038. Note that the name must be a simple filename and not a pathname. The actual DOS 3.3 filename must be placed at +039 (length) and +040 to +087 (name). (DOS 3.3 actually restricts filenames to 30 characters.)

- Set the ProDOS file type, auxiliary type and access to values which correspond to the DOS 3.3 file type:

DOS 3.3 File Type	ProDOS File Type	ProDOS Auxiliary Type	ProDOS Access
$00 (T)	$04	$0000	$E3

$80 (*T)	$04	$0000	$21
$01 (I)	$FA	$0C00	$E3
$81 (*I)	$FA	$0C00	$21
$02 (A)	$FC	*	$E3
$82 (*A)	$FC	*	$21
$04 (B)	$06	**	$E3
$84 (*B)	$06	**	$21
$08 (S)	$06	$0000	$E3
$88 (*S)	$06	$0000	$21
$10 (R)	$FE	$0000	$E3
$90 (*R)	$FE	$0000	$21
$20 (A)	$06	$0000	$E3
$A0 (*A)	$06	$0000	$E3
$40 (B)	$06	$0000	$E3
$C0 (*B)	$06	$0000	$21

* Set the auxiliary type for an A file to the memory address from which the program was saved. This is usually $0801.
** Set the auxiliary type for a B file to the value stored in the first two bytes of the the file (this is the default load address).

- Set the storage type code to $01.
- Set the size of file in blocks, date of creation, date of modification, time of creation and time of modification all to $0000.
- Set the end-of-file position to the length of the DOS 3.3 file, in bytes. For a B file (code $04 or $84), this number is stored in the third and fourth bytes of the file. For an I file (code $01 or $81) or an A file (code $02 or $82), this number is stored in the first and second bytes of the file.
- Set the operating system type to $01.
- Set the native file type code to the value of the DOS 3.3 file type code.

Attribute bytes inside a DOS 3.3 file (if any) must not be included in the data portion of the Binary II file. This includes the first four bytes of a B (Binary) file, and the first two bytes of an A (AppleSoft) or I (Integer BASIC) file.

Further Reference

- GS/OS Reference
- ProDOS 8 Technical Reference Manual
- Apple II File Type Note, File Type $E0, Auxiliary Type $0001
- Apple II File Type Note, File Type $E0, Auxiliary Type $8002
- Apple II Miscellaneous Technical Note #14, Guidelines for Telecommunication Programs

END OF FILE FTN.E0.8000

```
##########################################################################
### FILE: FTN.E0.8002
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: **$E0 (224)**
Auxiliary Type: **$8002**

Full Name: **NuFile Exchange Archival Library**
Short Name: **ShrinkIt (NuFX) document**

Revised by: **Andy Nicholas and Matt Deatherage** **July 1990**
Written by: **Matt Deatherage** **July 1989**

Files of this type and auxiliary type contain NuFX Archival Libraries. Changes since July 1989: Rewrote major portions to reflect Master Version $0002 of the NuFX standard.

Introduction

NuFX is a robust, full-featured archival standard for the Apple II family. The standard, as presented in this Note, allows for full archival of ProDOS and GS/OS files while keeping all file attributes with each file, as well as providing necessary archival functions such as multiple compression schemes and multiple archival implementations of the same standard. NuFX is implemented in the application ShrinkIt, a free archival utility program for enhanced IIe, IIc and IIgs computers. (Versions for earlier Apple II models are also available.)

The NuFX standard was developed by Andrew Nicholas for Paper Bag Productions. Comments or suggestions on the NuFX standard, or comments and suggestions on ShrinkIt are welcome at:

Paper Bag Productions
8415 Thornberry Drive East
Upper Marlboro, MD 20772
Attn: NuFX Technical Support
America Online: ShrinkIt
GEnie: ShrinkIt
CompuServe: 70771,2615

History

The Apple II community has always lacked a well-defined method for archiving files. NuFX is an attempt to rectify the situation by providing a flexible, consistent standard for archiving files, disks, and other computer media. Although many files are archived using the Binary II standard (see Apple II File Type Note, File Type $E0, Auxiliary Type $8000), it was not designed as an archival standard and its continued use as such creates problems. More people are using Binary II as an archival standard than as a way to keep attributes with a file when transferred, and this use is causing the original

intent of Binary II to become lost and unused.

NuFX, developed as an archival standard for the days of GS/OS, allows:

- Filenames longer than 64 characters (GS/OS can create 8,000-character filenames).
- A convenient way to add to, remove from, and work on an archive.
- Including GS/OS files which contain resource forks.
- Including entire disk images.
- Including comments with a file.
- A convenient way to represent a file compressed or encrypted by a specific application.
- A true archive standard. Binary IIs original intent was to make transfer of Apple II files from local machines to large information services possible; otherwise, a file's attribute information would be lost. Use of Binary II to archive files rather than simply maintain their attributes stretches it beyond its original intent.

Adding all of these features to the existing Binary II standard would be nearly impossible without violating the existing standard and causing a great deal of confusion. Although Binary II is flexible, it is simply unable to address all of these concerns without alienating existing Binary II extraction programs.

To provide some differentiation between standards and provide a better functioning format, this Note presents a new standard called NuFX (NuFile eXchange for the Apple II; pronounced new-F-X). NuFX fixes the problems that Apple IIgs users would soon be experiencing as other filing systems become available for GS/OS. NuFX attempts to stem a set of problems before they have a chance to develop. NuFX provides all of the features of Binary II, but goes further to allow the user the ultimate in flexibility, usefulness and performance.

Additional Date/Time Data type:

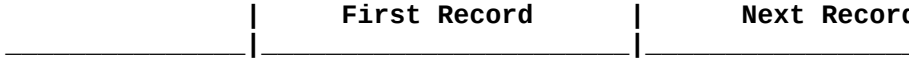
Date/Time (8 Bytes):

+000	second	Byte	The second, 0 through 59.
+001	minute	Byte	The minute, 0 through 59.
+002	hour	Byte	The hour, 0 through 23.
+003	year	Byte	The current year minus 1900.
+004	day	Byte	The day, 0 through 30.
+005	month	Byte	The month, 0 through 11 (0 = January).
+006	filler	Byte	Reserved, must be zero.
+007	weekDay	Byte	The day of the week, 1 through 7 (1 = Sunday).

The format of the Date/Time field is identical to that described for the ReadTimeHex call in the Apple IIgs Toolbox Reference Manual.

Implementation

Figure 1 illustrates the basic structure of a NuFX archive.



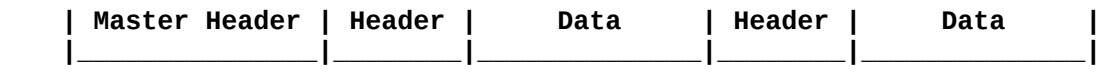


Figure 1-NuFX Archive Structure

A single master header block contains values which describe the entire archive (those with knowledge of structured programming may consider them archive globals). Each of the succeeding header blocks contains only information about the record it precedes (consider each an archive local).

Each header block is followed by a list of threads, which is followed by the actual threads. The data for each thread may be a data fork, resource fork, message, control sequence for a NuFX utility program, or almost any kind of sequential data.

Possible Block Combinations:

The blocks must occur in the following fashion:

```
Master Header Block containing N entries

Header Block
Threads list:
    filename_thread (16 bytes)
    message_thread (16 bytes)
    data thread (16 bytes)
    .
    .
    .
filename_thread's data (filename_thread's comp_thread_eof # of bytes)
message_thread's data (message_thread's comp_thread_eof # of bytes)
data_thread's data (data_thread's comp_thread_eof # of bytes)
.
.
.
Next Header Block (notice no second Master Header block)
Threads list (message, control, data or resource)
.
.
.
Nth Header Block
Threads list (message, control, data or resource)
```

Master Header Block Contents

+000	nufile_id	6 Bytes	These six bytes spell the word "NuFile" in alternating ASCII (low, then high) for uniqueness. The six bytes are $4E $F5 $46 $E9 $6C $E5.
+006	master_crc	Word	A 16-bit cyclic redundancy check (CRC) of the remaining fields in this block (bytes +008 through +047). Any programs which modify the master header block must recalculate the CRC for the master header. (see the section "A Sample CRC Algorithm") The initial value of this CRC is $0000.
+008	total_records	Long	The total number of records in this

			archive file. It is possible to chain multiple records (files or disks) together, as it is possible to chain different types of records together (mixed files and disks).
+012	archive_create_when	Date/Time	The date and time on which this archive was initially created. This field should never be changed once initially written. If the date is not known, or is unable to be calculated, this field should be set to zero. If the weekday is not known, or is unable to be calculated, this field should be set to null.
+020	archive_mod_when	Date/Time	The date of the last modification to this archive. This field should be changed every time a change is made to any of the records in the archive. If the date is not known, or is unable to be calculated, this field should be set to zero. If the weekday is not known, or is unable to be calculated, this field should be set to null.
+028	master_version	Word	The master version number of the NuFX archive. This Note describes master_version $0002, for which the next eight bytes are zeroed.
+030	reserved	8 Bytes	Must be null ($00000000).
+038	master_eof	Long	The length of the NuFX archive, in bytes. Any programs which modify the length of an archive, either increasing it or decreasing it in size, must change this field in the master header to reflect the new size.

Header Block Contents:

Following the Master Header block is a regular Header Block, which precedes each record within the NuFX archive. A cyclic redundancy check (CRC) has been provided to detect archives which have possibly been corrupted. The only time the CRC should be included in a block is for the Master Header and for each of the regular Header Blocks. The CRC ensures reliability and data integrity.

+000	nufx_id	4 Bytes	These four bytes spell the word "NuFX" in alternating ASCII (low, then high) for uniqueness. The four bytes are $4E $F5 $46 $D8.
+004	header_crc	Word	The 16-bit CRC of the remaining fields of this block (bytes +006 through the end of the header block and any threads following it). This field is used to verify the integrity of the rest of the block. Programs which create NuFX archives must include this in every header. It is up to the discretion of the extracting program to check the validity

			of this CRC. Any programs which might modify the header of a particular record must recalculate the CRC for the header block. The initial value for this CRC is zero ($0000).
+006	attrib_count	Word	This field describes the length of the attribute section of each record in bytes. This count measures the distance in bytes from the first field (offset +000) up to and including the filename_length field. By convention, the filename_length field will always be the last 2 bytes of the attribute section regardless of what has preceded it.
+008	version_number	Word	Version of this record. If version_number is $0000, no option_list fields are present. If the version_number is $0001 option_list fields may be present. If the version_number is $0002 then option_list fields may be present and a valid CRC-16 exists for the compressed data in the data threads of this record. If the version_number is $0003 then option_list fields may be present and a valid CRC-16 exists for the uncompressed data in the data threads of this record. The current version number is $0003 and should always be used when making archives.
+010	total_threads	Long	The number of thread subrecords which should be expected immediately following the filename or pathname at the end of this header block. This field is extremely important as it contains the information about the length of the last third of the header.
+014	file_sys_id	Word	The native file system identifier: $0000 reserved $0001 ProDOS/SOS $0002 DOS 3.3 $0003 DOS 3.2 $0004 Apple II Pascal $0005 Macintosh HFS $0006 Macintosh MFS $0007 Lisa File System $0008 Apple CP/M $0009 reserved, do not use (The GS/OS Character FST returns this value) $000A MS-DOS $000B High Sierra $000C ISO 9660 $000D AppleShare $000E-$FFFF Reserved, do not use If the file system of a disk being archived is not known, it should be set to zero.
+016	file_sys_info	Word	Information about the current filing

			system. The low byte of this word (offset +016) is the native file system separator. For ProDOS, this is the slash (/ or $2F). For HFS and GS/OS, the colon (: or $3F) is used, and for MS-DOS, the separator is the backslash (\ or $5C). This separator is provided so archival utilities may know how to parse a valid file or pathname from the filename field for the receiving file. GS/OS archival utilities should not attempt to parse pathnames, as it is not possible to build in syntax rules for file systems not currently defined. Instead, pass the pathname directory to GS/OS and attempt translation (asking the user for suggestions) only if GS/OS returns an "Invalid Path Name Syntax" error. The high byte of this word is reserved and should remain zero.
+018	access	Flag Long	Bits 31-8 reserved, must be zero Bit 7 (D) 1 = destroy enabled Bit 6 (R) 1 = rename enabled Bit 5 (B) 1 = file needs to be backed up Bits 4-3 reserved, must be zero Bit 2 (I) 1 = file is invisible Bit 1 (W) 1 = write enabled Bit 0 (R) 1 = read enabled
+022	file_type	Long	The file type of the file being archived. For ProDOS 8 or GS/OS, this field should always be what the operating system returns when asked. For disks being archived, this field should be zero.
+026	extra_type	Long	The auxiliary type of the file being archived. For ProDOS 8 or GS/OS, this field should always be what the operating system returns when asked. For disks being archived, this field should be the total number of blocks on the disk.
+030	storage_type	Word	For Files: The storage type of the file. Types $1 through $3 are standard (one-forked) files, type $5 is an extended (two-forked) file, and type $D is a subdirectory.
	file_sys_block_size	Word	For Disks: The block size used by the device should be placed in this field. For example, under ProDOS, this field will be 512, while for HFS it might be 524. The GS/OS Volume call will return this information if asked.
+032	create_when	Date/Time	The date and time on which this record was initially created. If the creation date and time are available from a disk device, this information should be included. If the date is not known, or is unable to be calculated, this field should be set to zero. If the weekday is not

			known, or is unable to be calculated, this field should be set to zero.
+040	mod_when	Date/Time	The date and time on which this record was last modified. If the modification date and time are available from a disk device, this information should be included. If the date is not known, or is unable to be calculated, this field should be set to zero. If the weekday is not known, or is unable to be calculated, this field should be set to zero.
+048	archive_when	Date/Time	The date and time on which this record was placed in this archive. If the date is not known, or is unable to be calculated, this field should be set to zero. If the weekday is not known, or is unable to be calculated, this field should be set to zero.

The following option_list information is only present if the NuFX version number for this record is $0001 or greater.

+056	option_size	Word	The length of the FST-specific portion of a GS/OS option_list returned by GS/OS. This field may be $0000, indicating the absence of a valid option_list.

A GS/OS option_list is formatted as follows:

+000	buffer_size	
	Word	Size of the buffer for GS/OS to place the option_list in, including this count word. This must be at least $2E.
+002	list_size	
	Word	The number of bytes of information returned by GS/OS.
+004	file_sys_ID	
	Word	A file system ID word (see list above) identifying the FST owning the file in question.
+006	option_bytes	
	Bytes	The bytes returned by the FST. There are (buffer_size - 6) of them.

The option_list contains information specific to native file systems that GS/OS doesn't normally use (such as true creator_type, file_type, and access privileges for AppleShare). Other FSTs released in the future will follow similar conventions to return native file system specific parameters in the option_list. Information in the option_list should always be copied from file to file.

The value option_size in the NuFX header is the value of list_size minus two. Immediately following the option_size count word are (list_size - 2) bytes. To pass these values back to the destination file system, construct an option_list with a suitably large buffer_size, a list_size of the NuFX option_size + 2, the file_sys_id of the source file, and the FST-returned

option_bytes.

+058	list_bytes	Bytes	FST-specific bytes returned in an option_list. These are the bytes in the GS/OS option_list not including the FST ID word. There are option_size of them. If option_size is an odd number, one zero byte of padding is added to keep the block size an even number.

Because the attributes section does not have a fixed size, the next field must be found by looking two bytes before the offset indicated by attrib_count (+006).

+attrib_count - 2			
	filename_length		
		Word	Obsolete, should be set to zero. In previous versions of NuFX, this field was the length of a file name or pathname immediately following this field. To allow the inclusion of future additional parameters in the attributes section, NuFX utility programs should rely on the attribs_count field to find the filename_length field. Current convention is to zero this field when building an archive and put the file or pathname into a filename thread so the record can be renamed in the archive. Archival programs should recognize both methods to find a valid file name or pathname.
+attrib_count			
	filename	Bytes	Filename or partial pathname if applicable. If this is a disk being archived, then the volume_name should be included in this field. If a volume name is included in this field, a separator should not be included in, or precede the name. If a volume name is not available, then this field should be zeros. If a partial pathname is specified, the directories to which the current pathname refers need not have preceded this particular record. The extraction program must test each referenced directory individually. If the directory in question does not exist, the extracting program should create it. Any utility which extracts file from a NuFX archive must not assume that this field will be in a format it is able to handle. In particular, extraction programs should check for syntax

unacceptable to the operating system under which they run and perform whatever conversions are necessary to parse a legal filename or pathname. In general, assume nothing. (GS/OS programs should pass the filename or pathname directly to GS/OS, and only attempt to convert the name if GS/OS returns an "invalid pathname syntax" error.)

Both high and low ASCII values are valid but may not mean the same to each file system (for example, all eight bits are significant in AppleShare pathnames while only seven are significant in ProDOS pathnames).

Threads

Thread Records are 16-byte records which immediately follow the Header Block (composed of the attributes and file name of the current record) and describe the types of data structures which are included with a given record. The number of Thread Records is described in the attribute section by a Word, total_threads.

Each Thread Record should be checked for the type of information that a given utility program can extract. If a utility is incapable of extracting a particular thread, that thread should be skipped (with the exception of extended files under ProDOS 8, which should be dearchived into AppleSingle format, or both threads should be skipped). If a utility finds a redundancy in a Thread Record, it must decide whether to skip the record or to do something with that particular thread (i.e., if a utility finds two message_thread threads it can either ignore the second one or display it. Likewise, if a utility finds two data_thread threads for the same file, it should inspect the thread_kind of each. If they match, it can either overwrite the first thread extracted, or warn the user and skip the second thread).

Thread records can be represented as follows:

+000	thread_class	Word	The classification of the thread: $0000 message_thread $0001 control_thread $0002 data_thread $0003 filename_thread
+002	thread_format	Word	The format of the data within the thread: $0000 Uncompressed $0001 Huffman Squeeze $0002 Dynamic LZW/1 (ShrinkIt specific) $0003 Dynamic LZW/2 (ShrinkIt specific) $0004 Unix 12-bit Compress $0005 Unix 16-bit Compress
+004	thread_kind	Word	Describes the kind of data within the thread.

thread_kind must be interpreted on the basis of thread_class. See the table below for the currently defined thread_kind interpretations:

	class $0000	class $0001	class $0002	class $0003
kind $0000	ASCII text	create directory	data fork of file	filename
kind $0001	see below	undefined	disk image	undefined
kind $0002	see below	undefined	resource fork of file	undefined

+006	thread_crc	Word	For version_number $0003, this field is the CRC of the original data before it was compressed or otherwise transformed. The CRC-16's initial value is set to $FFFF.
+008	thread_eof	Long	The length of the thread when uncompressed.
+012	comp_thread_eof	Long	The length of the thread when compressed.

Class $0000 with kind $0000 is obsolete and should not be used.

Class $0000 with kind $0001 has a predefined comp_thread_eof and a thread_eof whose length may change. This way, a certain amount of space may be allocated when a record is created and edited later.

Class $0000 with kind $0002 is a standard Apple IIgs icon. comp_thread_eof is the length of the icon image; thread_eof is ignored.

Class $0003 with kind $0000 has a predefined comp_thread_eof and a thread_eof whose length may change. After this record is placed into the archive, the thread_eof can be changed if the name is changed, but the length of the name may not extend beyond the space allocated for it, comp_thread_eof.

A thread_format of $0001 indicates Huffman Squeeze. NuFX's Huffman is the same Huffman used by ARC v5.x, SQ and USQ, the source of which is publicly available and was originally written by Richard Greenlaw. The first word of the thread data is the number of nodes followed by the Huffman tree and the actual data. This is also the same algorithm decoded by the Apple II version of USQ written by Don Elton. The C source to this is widely available.

A thread_format of $0002 indicates a special variant of LZW (LZW/1) used by ShrinkIt. The first two bytes of this thread are a CRC-16 of the uncompressed data within the thread. This CRC-16 is initialized to zero ($0000). The third byte is the low-level volume number used by the eight-bit version of ShrinkIt to format 5.25" disks. The fourth byte is the run-length character used to decode the rest of the thread. The data which comprises the compressed file or disk immediately follows the RLE character.

When ShrinkIt compresses a file, it reads 4096-byte chunks of the file until it reaches the file's EOF. The last 4096-byte chunk is padded with zeroes if the file's length is not an exact multiple of 4096. Compressing a disk is also done by reading sequential blocks of 4096-bytes.

Each 4K chunk is first compressed with RLE compression. The RLE character is determined by reading the fourth byte of the thread. The RLE character which is used by most current versions of ShrinkIt is $DB. A run of characters is represented by three bytes, consisting of the run character, the number of characters in the run and the character in the run. If the 4K chunk expands after being compressed with RLE then the uncompressed 4K chunk is passed to the LZW compressor. If the 4K chunk shrinks after being compressed with RLE

then the RLE-compressed image of the 4K chunk is passed to the LZW compressor.

ShrinkIt's LZW compressor individually compresses each 4K chunk passed to it by using variable length (9 to 12 bits) codes. The way that ShrinkIt's LZW compressor functions is almost identical to the algorithm used in the public domain utility Compress. The first code is $0101. The LZW string table is cleared before compressing each 4K chunk. If the compressed chunk increases in size, then the previous 4K chunk (which may be run-length-encoded or just uncompressed data) is written to the file.

The first word of every 4K chunk is aligned to a byte boundary within the file and is the length which resulted from the attempt at compressing the chunk with RLE. If the value of this word is 4096, then RLE was not successful at compressing the chunk. A single byte follows the word and indicates whether or not LZW was performed on this chunk. A value of zero indicates that LZW was not used, while a value of one indicates that LZW was used and that the chunk must first be decompressed with LZW before doing any further processing.

To decompress a file, each 4K chunk must first be expanded if it was compressed by LZW. If the 4K chunk wasn't compressed with LZW, then the word which appears at the beginning of each chunk must be used to determine if the data for the current chunk needs to be processed by the run-length decoder. If the value of the word is 4096, then run-length decoding does not need to occur because the data is uncompressed.

If the word indicates that the length of the chunk after being decompressed by LZW is 4096-bytes long, then no run-length decoding needs to take place. If value of the word is less than 4096 then the chunk must be run-length decoded to 4096 bytes.

There are four varying degrees of compression which can occur with a chunk: it can be uncompressed data. It can be run-length-encoded data without LZW compression. It can also be uncompressed data on which RLE was attempted (but failed) and then was subsequently compressed with LZW. Or, finally, the chunk can be compressed with RLE and then also compressed with LZW.

A thread_format of $0003 indicates a special variant of LZW (LZW/2) used by ShrinkIt. The first byte is the low-level volume number used by the eight-bit version of ShrinkIt to format 5.25" disks. The second byte is the run-length character used to decode the rest of the thread. The data which comprises the compressed file or disk immediately follows the second byte of the thread.

The format of LZW/2 is almost the same as LZW/1 with a few exceptions. Unlike LZW/1, where the LZW string table is automatically cleared before each 4K chunk is processed, the LZW string table used by LZW/2 is only cleared when the table becomes full, indicating a change in the redundancy of the source text. Not clearing the string table almost always yields improved compression ratios because the compressor's dictionary is not being depleted every 4K and larger strings are allowed to accumulate. The clear code used by ShrinkIt is $100. Whenever the decompressor sees a $100 code, it must clear the string table.

The string table is also cleared when the compressor has to "back track" because a 4K chunk became larger. Whenever a chunk that is not compressed by LZW is seen by the decompressor, the LZW string table must be cleared. Bits 0-12 of the first word of each chunk in a LZW/2 thread indicate the size of the chunk after being compressed with RLE. The high bit (bit 15) indicates whether or not LZW was used on the chunk. If LZW was not used (bit 15 = 0),

the data for the chunk immediately follows the first word. If LZW was used (bit 15 = 1), a second word which is a count of the total number of bytes used by the current chunk follows the first word. The mark of the next chunk can be found by taking the mark at the beginning of the current chunk and adding the second word to it, using that as an offset for a ProDOS 8 or GS/OS SetMark call. This is not normally necessary because the next chunk is processed immediately after the current chunk.

This second word is an improvement over LZW/1 because if a chunk becomes corrupted, but the second word is valid, the next chunk can be found and most of the file recovered. The second word is not needed (and not present) when LZW is not used on the chunk because the first word is also a count of the number of bytes which follow that word.

A thread_format of $0004 indicates that a maximum of 12 bits per LZW code by Compress was used to build this thread. The actual thread data contains Compress's usual three-byte signature, the third byte of which contains the actual number of bits per LZW code that was actually used. The number of bits may be less than or equal to 12. Optimally, this requires (at 12 bits) a 16K hash table to decode and should be used only for transferring to machines with limited amounts of memory. The C source to Compress is in the public domain and is widely available.

A thread_format of $0005 indicates that a maximum of 16 bits per LZW code by Compress was used to build this thread. The actual thread data contains Compress's usual three-byte signature, the third byte of which contains the actual number of bits per LZW code that was actually used. The number of bits may be less than or equal to 16. Optimally, this requires (at 16 bits) a 256K hash table to decode. The C source to Compress is in the public domain and is widely available.

If a control_thread indicates that a directory should be created on the destination device, the path to be created must take the form of a ProDOS partial pathname. That is, the path must not be preceded with a volume name. For example, /Stuff/SubDir is an invalid path for this control_thread, while SubDir/AnotherSubDir is valid.

If a control_thread indicates that a path is to be created, all subdirectories that are contained in the pathname must be created.

control_thread threads will eventually be used to control the execution of utility programs by allowing them to create, rename, and delete directories and files and to move and modify files. A form of scripting language will eventually be able to allow utility programs to perform these actions automatically. control_thread threads will allow extraction programs to perform operations similar to those of the Apple IIgs Installer, allowing updates to program sets dependent on such things as creation or modification dates and version numbers.

Extra Information

If the file system of a particular disk is not known, the file_sys_id field should be set to zero, the volume name should also be zeroed, and all the other fields pertaining only to files should be set to zero.

If the file system of a particular disk is known, as many of the fields as possible should be filled with the correct information. Fields which do not

pertain to an archived disk should remain set to zero.

If an entire disk is added to the archive without some form of compression (i.e., record_format = uncompressed), then the blocks which comprise the disk image must be added sequentially from the first through the last block. Since there will be no character included in the data stream to mark the end or beginning of a block, extraction programs should rely on the file_sys_block_size field to determine how many bytes to read from the record to properly fill a block.

Some Useful Thread Algorithms:

The beginning of the thread records can be found with the following algorithm:

```
Threads := (mark at beginning of header) + (attrib_count) +
           (filename_length)
```

The end of the thread records can be found with the following algorithm:

```
endOfThreads := Threads + (16 * total_threads)
```

The beginning of a data_thread can be found with the following formula:

```
Data Mark := endOfThreads + (comp_thread_eof of all threads in the thread
             list which are not data prior to finding a data_thread)
```

The beginning of a resource_thread may be found with the following algorithm:

```
Resource Mark := endOfThreads + (comp_thread_eof of all threads in the
                 thread list which are not data prior to finding a
                 resource_thread)
```

The next record can be found using the following algorithm:

```
Next Mark := endOfThreads + (comp_thread_eof of each thread)
```

The file name and its length can be found with the following algorithm:

```
if (filename_length > 0)
    then
        length of filename is filename_length;
        filename is found at attrib_count;
    else
        look through list of threads for a filename_thread;
        if you find one, then length of filename is thread_eof;
        if you don't find one, then you don't have a filename.
```

Directories

Directories are handled almost the same way that normal files are handled with the exception that there will be no data in the thread which follows the entry. A Thread Record must exist to inform a utility that a directory is to be created through the use of the proper control_thread value.

Directories do not necessarily have to precede a record which references a directory. For example, if a record contains Stuff/MyStuff, the directory Stuff need not exist for the extracting program to properly extract the

record. The extracting program must check to see if each of the directories referenced exist, and if one does not exist, create it. While this method places a great burden on the abilities of the extraction program, it avoids the anomalies associated with the deletion of directories within an archive.

A Sample CRC Algorithm

Paper Bag Productions provides the source code to a very fast routine which does the CRC calculation as needed for NuFX archives. The routine makeLookup needs to be called only once. After the first call, the routine doByte should be called repeatedly with each new byte in succession to generate the cumulative CRC for the block. The CRC word should be reset to null ($0000) before beginning each new CRC.

This is the same CRC calculation which is done for CRC/Xmodem and Ymodem. The code is easily portable to a 16-bit environment like the Apple IIgs. The only detrimental factor with this routine is that it requires 512 bytes of main memory to operate. If you can spare the space, this is one of the fastest routines Paper Bag Productions knows to generate a CRC-16 on a 6502-type machine.

The CRC word should be reset to $0000 for normal CRC-16 and to $FFFF before generating the CRC on the unpacked data for each data thread.

```
*------------------------------
* fast crc routine based on table lookups by
* Andy Nicholas - 03/30/88 - 65C02 - easily portable to nmos 6502 also.
* easily portable into orca/m format, just snip and save.
* Modified for generic EDAsm type assemblers - MD 6/19/89

         X6502                          turn 65c02 opcodes on

*------------------------------
* routine to make the lookup tables
*------------------------------

makeLookup
         LDX   #0                       zero first page
zeroLoop STZ   crclo,x                  zero crc lo bytes
         STZ   crchi,x                  zero crc hi bytes
         INX
         BNE   zeroLoop

*------------------------------
* the following is the normal bitwise computation
* tweeked a little to work in the table-maker

docrc
         LDX   #0                       number to do crc for

fetch    TXA
         EOR   crchi,x                  add byte into high
         STA   crchi,x                  of crc

         LDY   #8                       do 8 bits
loop     ASL   crclo,x                  shift current crc-16 left
```

```
         ROL   crchi,x
         BCC   loop1

* if previous high bit wasn't set, then don't add crc
* polynomial ($1021) into the cumulative crc.  else add it.

         LDA   crchi,x                add hi part of crc poly into
         EOR   #$10                   cumulative crc hi
         STA   crchi,x

         LDA   crclo,x                add lo part of crc poly into
         EOR   #$21                   cumulative crc lo
         STA   crclo,x
loop1    DEY                          do next bit
         BNE   loop                   done? nope, loop

         INX                          do next number in series (0-255)
         BNE   fetch                  didn't roll over, so fetch more
         RTS                          done

crclo    ds    256                    space for low byte of crc table
crchi    ds    256                    space for high bytes of crc table


*------------------------------
* do a crc on 1 byte/fast
* on initial entry, CRC should be initialized to 0000
* on entry, A = byte to be included in CRC
* on exit, CRC = new CRC
*------------------------------

doByte
         EOR   crc+1                  add byte into crc hi byte
         TAX                          to make offset into tables

         LDA   crc                    get previous lo byte back
         EOR   crchi,x                add it to the proper table entry
         STA   crc+1                  save it

         LDA   crclo,x                get new lo byte
         STA   crc                    save it back

         RTS                          all done

crc      dw    0000                   cumulative crc for all data
```

The following CRC check is written in APW assembler format for an Apple IIgs with 16-bit memory and registers on entry.

```
crcByte  start

crc      equ         $0
crca     equ         $2
crcx     equ         $4
crctemp  equ         $6

         sta         crca                                          4
         stx         crcx                                          4
```

```
         eor         crc+1              on entry, number to add to CRC    4
         and         #$00ff             is in (A)                         3
         asl         a                                                    2
         tax                                                              2
         lda         crc16Table,x                                         5
         and         #$00ff                                               3
         sta         crcTemp                                              4

         lda         crc-1                                                4
         eor         crc16Table,x                                         5
         and         #$ff00                                               3
         ora         crcTemp                                              4
         sta         crc                                                  4

         lda         crca                                                 4
         ldx         crcx                                                 4
         rts                                                   cycles = 59


;
; CRC-16 Polynomial = $1021
;
crc16table anop
         dc    i'$0000, $1021, $2042, $3063, $4084, $50a5, $60c6, $70e7'
         dc    i'$8108, $9129, $a14a, $b16b, $c18c, $d1ad, $e1ce, $f1ef'
         dc    i'$1231, $0210, $3273, $2252, $52b5, $4294, $72f7, $62d6'
         dc    i'$9339, $8318, $b37b, $a35a, $d3bd, $c39c, $f3ff, $e3de'
         dc    i'$2462, $3443, $0420, $1401, $64e6, $74c7, $44a4, $5485'
         dc    i'$a56a, $b54b, $8528, $9509, $e5ee, $f5cf, $c5ac, $d58d'
         dc    i'$3653, $2672, $1611, $0630, $76d7, $66f6, $5695, $46b4'
         dc    i'$b75b, $a77a, $9719, $8738, $f7df, $e7fe, $d79d, $c7bc'
         dc    i'$48c4, $58e5, $6886, $78a7, $0840, $1861, $2802, $3823'
         dc    i'$c9cc, $d9ed, $e98e, $f9af, $8948, $9969, $a90a, $b92b'
         dc    i'$5af5, $4ad4, $7ab7, $6a96, $1a71, $0a50, $3a33, $2a12'
         dc    i'$dbfd, $cbdc, $fbbf, $eb9e, $9b79, $8b58, $bb3b, $ab1a'
         dc    i'$6ca6, $7c87, $4ce4, $5cc5, $2c22, $3c03, $0c60, $1c41'
         dc    i'$edae, $fd8f, $cdec, $ddcd, $ad2a, $bd0b, $8d68, $9d49'
         dc    i'$7e97, $6eb6, $5ed5, $4ef4, $3e13, $2e32, $1e51, $0e70'
         dc    i'$ff9f, $efbe, $dfdd, $cffc, $bf1b, $af3a, $9f59, $8f78'
         dc    i'$9188, $81a9, $b1ca, $a1eb, $d10c, $c12d, $f14e, $e16f'
         dc    i'$1080, $00a1, $30c2, $20e3, $5004, $4025, $7046, $6067'
         dc    i'$83b9, $9398, $a3fb, $b3da, $c33d, $d31c, $e37f, $f35e'
         dc    i'$02b1, $1290, $22f3, $32d2, $4235, $5214, $6277, $7256'
         dc    i'$b5ea, $a5cb, $95a8, $8589, $f56e, $e54f, $d52c, $c50d'
         dc    i'$34e2, $24c3, $14a0, $0481, $7466, $6447, $5424, $4405'
         dc    i'$a7db, $b7fa, $8799, $97b8, $e75f, $f77e, $c71d, $d73c'
         dc    i'$26d3, $36f2, $0691, $16b0, $6657, $7676, $4615, $5634'
         dc    i'$d94c, $c96d, $f90e, $e92f, $99c8, $89e9, $b98a, $a9ab'
         dc    i'$5844, $4865, $7806, $6827, $18c0, $08e1, $3882, $28a3'
         dc    i'$cb7d, $db5c, $eb3f, $fb1e, $8bf9, $9bd8, $abbb, $bb9a'
         dc    i'$4a75, $5a54, $6a37, $7a16, $0af1, $1ad0, $2ab3, $3a92'
         dc    i'$fd2e, $ed0f, $dd6c, $cd4d, $bdaa, $ad8b, $9de8, $8dc9'
         dc    i'$7c26, $6c07, $5c64, $4c45, $3ca2, $2c83, $1ce0, $0cc1'
         dc    i'$ef1f, $ff3e, $cf5d, $df7c, $af9b, $bfba, $8fd9, $9ff8'
         dc    i'$6e17, $7e36, $4e55, $5e74, $2e93, $3eb2, $0ed1, $1ef0'
         end
```

Further Reference

- **ProDOS 8 Technical Reference Manual**
- **GS/OS Reference**
- **Apple IIgs Toolbox Reference Manual**
- **Apple II File Type Note, File Type $E0, Auxiliary Type $8000**
- **Apple II Miscellaneous Technical Note #14, Guidelines for Telecommunication Programs**
- **"A Technique for High-Performance Data Compression," T. Welch, IEEE Computer, Vol. 17, No.6, June 1984, pp. 8-19.**

END OF FILE FTN.E0.8002

```
##########################################################################
### FILE: FTN.E0.8004
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: **$E0 (224)**
Auxiliary Type: **$8004**

Full Name: **Davex archived volume**
Short Name: **Davex archived volume**

Written by: **Dave Lyons** **May 1990**

Files of this type and auxiliary type contain an archived image of a ProDOS volume.

Davex is a ProDOS 8 command-line shell and program launcher compatible with all Apple II computers that can run ProDOS 8. It supports stuff like wildcards, command history, print spooling, sorted directory listings, and operations on whole directory structures. You can add your own assembly-language commands, too. Davex also allows (coincidentally) saving an image of any ProDOS volume into a file and restoring it later.

For more information on Davex, contact:

DAL Systems
P.O. Box 875
Cupertino, CA 95014
Attention: Davex Technical Support

File Structure

The first 512 bytes of a Davex archived volume are a header, described under "File Format" in this Note. After the header comes 512 bytes for each block on the saved volume, from zero on up. For blocks that are unused, you can just set the file mark ahead 512 bytes instead of writing 512 zero bytes--this way the unused blocks do not take up disk space, so the resulting file is only a few blocks larger than the number of used blocks on the original volume.

If you run out of room while you are creating an archived volume file, close the file and start another one with the same name on a new disk. The fileNumber field in the header is one in the first file, two in the second file, and so on.

This file format is suitable for ProDOS, but it is less useful on a file system (such as AppleShare) that does not allow for sparse files. A Davex archived volume file on an AppleShare server always takes up more blocks than the original volume contains.

File Format

identityCheck	(+000)	16 Bytes	These 16 bytes are required for historical reasons (there didn't used to be a special file type and auxiliary type to identify these files). The required value is $60 followed by "VSTORE [Davex]" and a $00. The characters have their high bits off.
fileFormat	(+016)	Byte	Must be $00. A nonzero value means the file format has changed in a way that isn't compatible with the current definition.
vstoreVers	(+017)	Byte	Version of Davex vstore command used to create this file (others use $00).
vrestoreVers	(+018)	Byte	Minimum version of the Davex vrestore command needed to read this file. Use $10 (version 1.0).
reserved	(+019)	13 Bytes	Reserved for future use.
deviceNum	(+032)	Byte	ProDOS 8 device number of the device that this file is a volume image from. Informational only.
totalBlocks	(+033)	Long	Number of blocks on the saved volume.
usedBlocks	(+037)	Long	Number of used blocks on the saved volume.
volumeName	(+041)	String	Name of the saved volume, with a leading length byte. This field is 16 bytes long. If the name is shorter than 15 characters, the remaining bytes are unused and should be zero.
reserved	(+057)	7 Bytes	Reserved for future use.
fileNumber	(+064)	Byte	This field contains one for the first file of an archive, n in the nth file (see above).
startingBlock	(+065)	Long	Block number corresponding to the data starting at offset (+512) in this file.
reserved	(+069)	443 Bytes	Reserved for future use.
theBlocks	(+512)	512*n Bytes	512 bytes of data for each block of the saved volume recorded in this file where n is the number of blocks.

Further Reference

- **ProDOS 8 Technical Reference**

END OF FILE FTN.E0.8004

```
##########################################################################
### FILE: FTN.E0.8006
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: **$E0 (224)**
Auxiliary Type: **$8006**

Full Name: **EZ Backup Saveset document**
Short Name: **EZ Backup Saveset document**

Written by: **Peter Easdown and Matt Deatherage** **September 1990**

Files of this type and auxiliary type contain savesets as produced by EZBackup.

EZ Backup is a backup utility that runs under GS/OS. It provides the ability to backup to either a removable block device or to files under any available file system.

For more information on EZ Backup, contact:

EZ-Soft Pty. Ltd.
G.P.O. Box 880
Sydney, N.S.W., 2001
Australia
Telephone: 011.61.2.365.1271
AppleLink: AUST0367

Definitions

The following definition is used in this document in addition to those defined for all Apple II file types:

Date/Time **An eight-byte date/time record as used by GS/OS and the Miscellaneous Tools.**

The File Format

EZ Backup savesets are divided into three major segments: the header, the file list and the data.

The Header

The header contains information relating to the saveset as a whole. It is a fixed length of 1,024 ($400) bytes and is defined as follows:

backupDateTime (+000) Date/Time
The date and time on which the backup took place.

fileCount (+008) Word
The number of files stored in the saveset.

rootDirectory (+010) 512 Bytes
A GS/OS input string containing the full pathname of the top-level directory of the saveset. Note that normally this is the volume name of the volume that was backed up.

fileList (+522) Long
A pointer used at run-time to point to the first entry of the file list. This field need not be zeroed when writing to disk.

majRelease (+526) Word
The major release component of the version number of EZ Backup used to create the saveset, as a two-byte integer.

minRelease (+528) Word
The minor release component of the version number of EZ Backup used to create the saveset, as a two-byte integer.

fileSysID (+530) Word
The file system ID of the volume that was backed up.

backupType (+532) Boolean Word
A flag that indicates whether the backup was a full backup or an incremental (changes only) backup. A value of TRUE means the backup is incremental.

selected (+534) Boolean Word
A run-time flag used to indicate that all files in the directory tree are selected. A value of TRUE indicates that all files are selected.

devIcon (+536) Long
A number indicating which icon to display for the root level when restoring. This is supplied in case the configuration of the machine doing the restore differs from that of the machine that made the backup. Following are the icon numbers and device types they represent:

File Server	$FFF5
CD-ROM	$FFF8
5.25" Drive	$FFF9
RAM Disk	$FFFA
3.5" Disk	$FFFB
5.25" Disk	$FFFC
Hard Disk	$FFFD

fileListLen	(+540)	Long	The length in bytes of the file list (defined later).
totalDisks	(+544)	Long	The number of disks required by the saveset (excluding the disk containing the file list). Note that this is not relevant to a file-based backup since a saveset file cannot be larger than the disk on which it resides.
reserved	(+548)	Word	Reserved for future use.
backupLen	(+550)	Long	The total size in bytes of the backup, including the header, the file list and the data.
reserved	(+554)	470 Bytes	Reserved for future use.

The File List

The file list contains a variable number of 128-byte records, the number of which is given by the fileCount field in the header. Each file list entry is defined as follows:

nextFile	(+000)	Long	A pointer that is used at run-time to point to the next file in the file list. Although only used a run-time, this field is useful in reconstructing the directory hierarchy when restoring savesets.
fileInfo	(+004)	62 Bytes	A GS/OS GetDirEntry parameter block that describes this file.
dataOffset	(+066)	Long	An offset in bytes from the beginning of the file that points to the location within the saveset at which the data fork (if any) is stored. This field is zero if there is no data fork.
resOffset	(+070)	Long	An offset in bytes from the beginning of the file that points to the location within the saveset at which the resource fork (if any) is stored. This field is zero if there is no resource fork.
optionListOffset	(+074)	Long	An offset in bytes from the beginning of the file that points to the location within the saveset at which the GS/OS option_list (if any) was stored. This field is zero if there is no option_list. Note that EZ Backup does not currently store the option_list for any ProDOS files.
optionListLen	(+078)	Word	The length in bytes of the option_list (if any).

parentFile	(+080)	Long	A pointer that contains the run-time address of the file list record of the directory that is a parent to the file described by this record. Although only used a run-time, this field is useful in reconstructing the directory hierarchy when restoring savesets.
currentDir	(+084)	Long	If the file described by this record is a directory, this field is a pointer to itself. This is supplied primarily for the restore operation so that the directory hierarchy may be rebuilt.
selected	(+088)	Word	Used at run-time to indicate a file record selection mode. If this field is zero, then some error occurred during the backup that prevented the file from being backed up. A non-zero value indicates that the file was correctly included in the saveset; only attempt to restore files that have a non-zero selection mode.
created	(+090)	Boolean Word	A flag used at restore time to indicate whether the file was created successfully. A value of TRUE means yes while FALSE means no.
fileName	(+092)	36 Bytes	A GS/OS output string containing the name of the file.

The Data

The data component of the file contains the contents of all of the files described in the file list in a contiguous stream of bytes. Each file begins at a 512-byte boundary. Each file list record takes 128 bytes, and any remaining bytes in the last 512-byte block of the file list are unused. Similarly, all stored data forks, resource forks, and option_lists start on 512-byte boundaries, and any remaining bytes in their last 512-byte blocks are unused.

Further Reference

- GS/OS Reference

END OF FILE FTN.E0.8006

```
##########################################################################
### FILE: FTN.E0.800A
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $E0 (224)
Auxiliary Type: $800A

Full Name: Replicator document
Short Name: Replicator document

Written by: Josef W. Wankerl & Matt Deatherage May 1992

Files of this type and auxiliary type contain images for the disk duplicating application Replicator.

Replicator is a commerical, desktop-based disk duplicating application available from GS+ Magazine.

For more information on Replicator or GS+ Magazine, contact:

GS+ Magazine
P.O. Box 15366
Chattanooga, TN 37415-0366
Attention: Replicator Technical Support
(615) 843-3988

America Online: GSPlusDiz
Delphi: GSPlusDiz
GEnie: JWANKERL
Internet: jwankerl@pro-gonzo.cts.com

FILE FORMAT

A Replicator file is an extended file with an empty data fork. The resource fork should contain the following resources:

Res Type	Resource ID	Contents	Description
$0001	$00000001	Bytes	The disk image, as read with a DRead GS/OS call.
$0002	$00000001	Long	The block count of the disk, as returned by a GS/OS Volume call.
$0003	$00000001	Word	The block size of the disk, as returned by a GS/OS Volume call.
$0004	$00000001	Word	Corresponds to the state of the Replicator disk window's "Number of copies" radio buttons. If this word is zero, the "Mass copy" radio button is selected; otherwise, the "Number of Copies" radio button is selected.

$8006 rPString	$00000001	String	Name of the disk volume as returned by a GS/OS Volume call, with spaces on both sides of the name, as Replicator uses this string in a window title.
$8006 rPString	$00000002	String	Name of the file system the disk was formatted with. You can use the fileSysID GS/OS returns to match the name of the file system to those returned by GetFSTInfo.
$8006 rPString	$00000003	String	Textual representation of the block count.
$8006 rPString	$00000004	String	Textual representation of the block size.
$8016 rText	$00000001	Bytes	The number of copies to make. Replicator inserts this text into the "number of copies" Line Edit control in the disk window.
$8029 rVersion	$00000001	Bytes	The minimum version of Replicator necessary to read this document. The only defined version is 1.0. The non-version fields of the rVersion resource should be set so the Finder displays "Requires Replicator <version>".
$802A rComment	$00000001	Bytes	Any comments to place in the "Comments" TextEdit box in the disk window, and also shown by the Finder in "Icon Info." This resource is optional.

As an example, a Replicator document of a 128K ProDOS RAM disk named ":RAM5", set to make five copies with "Mass copy" turned off and containing no comments would contain the following resources:

Res Type	Resource ID	Content Description
$0001	$00000001	$00020000 (131072) bytes of disk image data.
$0002	$00000001	$00000100 (256 blocks)
$0003	$00000001	$0200 (512 bytes per block)
$0004	$00000001	$0001 (Mass copy turned off)
$8006 (rPString)	$00000001	" :RAM5 "
$8006 (rPString)	$00000002	"ProDOS"
$8006 (rPString)	$00000003	"256"
$8006 (rPString)	$00000004	"512"
$8016 (rText)	$00000001	"5"
$8029 (rVersion)	$00000001	1.0 (release)

Further Reference

- GS/OS Reference
- Apple IIgs Technical Note #76, Miscellaneous Resource Formats

END OF FILE FTN.E0.800A

```
##########################################################################
### FILE: FTN.E2.FFFF
##########################################################################
```

Apple II
File Type Notes

Developer Technical Support

File Type: $E2 (226)
Auxiliary Type: $FFFF

Full Name: EasyMount document
Short Name: EasyMount document

Written by: Dave Lyons **May 1992**

Files of this type and auxiliary type contain EasyMount documents used by the System 6.0 EasyMount Finder extension.

The EasyMount Finder extension in System 6.0 creates EasyMount documents and uses them to let the user quickly connect to shared disks.

THE FILE FORMAT

An EasyMount document has the following format. In future versions, more data may be added to the end.

serverName	(+000)	String	Name of server to connect to.
entityName	(+xxx)	String	Always "AFPServer". Immediately follows serverName.
zoneName	(+xxx)	String	Name of zone containing server. Immediately follows entityName.
volumeName	(+097)	28 Bytes	Name String of specific volume to use on the server. This field takes 28 bytes regardless of the length of the String.
userName	(+125)	32 Bytes	User name String to connect with. This field takes 32 bytes regardless of the length of the String.
serverPassword	(+157)	8 Bytes	Password to connect with (padded with trailing zero bytes; all zeros to connect as Guest).
volumePassword	(+165)	8 Bytes	Volume password (padded with trailing zero bytes). Most servers do not use volume passwords, and EasyMount always fills this field with zeros.

Note that the userName field is big enough for a 31-character user name (with length byte). Although 32-character user names are valid under AppleShare, EasyMount can't deal with them (for historical reasons).

Passwords are stored UNENCRYPTED in EasyMount documents; finding one with a saved password is the same as finding that user's password to the server. Please take appropriate security precautions if you save passwords in EasyMount documents.

Further Reference

- **System 6.0 Documentation**

END OF FILE FTN.E2.FFFF

F I N I S