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 it's 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.

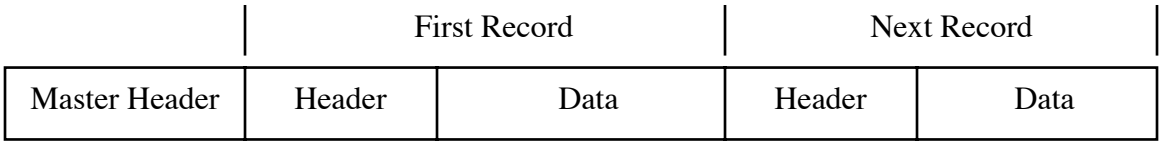

Figure 1–NuFX Archive Structure

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 Compssion," T. Welch, *IEEE Computer*, Vol. 17, No.6, June 1984, pp. 8-19.