APL64: Using Files

Basic Information

This chapter describing file functions applies only to files that you use system functions to read and write. It does not apply to workspace files, which have somewhat different rules.

APL64 allows you to store and access data in disk files independent of an APL workspace. This chapter explains the different types of data files that APL64 uses and the various functions you can use to create and manipulate these files. Using these file capabilities, you can:

- store more data than a workspace can hold and retrieve them in manageable portions
- perform file operations under program control
- share data and control access to files in a multiple user environment
- provide an interface between APL and non-APL programs.

There is a new component file system for APL64. It is called the colossal component file system. The colossal component file system is very similar to the traditional component file system (i.e., `⎕XFCREATE`, `⎕XFTIE`, and `⎕FAPPEND`) with several key advantages. They are:

- the ability to create component files as large as (2*63)-1 bytes (9 exabytes)
- the ability to create component files with (2*32)-1 components (over 4 billion)
- better re-use of empty space in the component file (i.e., relief for the "exploding freplace" problem)
- finer control of file synchronization
- improved file reliability, integrity and recovery

See the "Colossal Component File System" section later in this manual for more information.

Contents

Understanding the Basics

APL64 recognizes three types of files: colossal component files, traditional component files and native files. An APL component file is a sequence of APL arrays stored on disk. Each array can be of any datatype, rank, shape, or size. One can be a table of floating point numbers, while the next can be a four-page memo. A component can also contain the representation of a function, stored as a character vector.

Component files have an internal organization maintained by the APL system itself; they consist of a sequence of individually accessible and replaceable APL values. Regardless of the nature or size of the array, you refer to it by a single component number and retrieve it by that number. When you retrieve a component, the system automatically recognizes and handles the array's external structure and internal organization (the number of bytes per element and the interpretation of the arrangement of the bits). Component files are usable only by APL64 and APL+ systems.

Native files, by contrast, are theoretically usable by any application that can use the host operating system. APL64 treats native files as simple sequences of bytes, so when you write such a file, you must determine the organization and representation of the data. This represents maximum flexibility at the cost of higher programming effort. The program performing the retrieval must deal with where to start reading the material, how far to read, and whether and how to reshape the data.

There are three stages in the use of either type of file by an APL program. First, the program connects itself to the file by executing a system function. The function requires two arguments: a file identifier that the operating system recognizes to locate the file and an arbitrary number. APL64 performs the necessary interaction with the file system to allow access (known as "opening" the file) and establishes the number as the means by which to identify the file for subsequent operations within APL. The connection is known as a file tie, and the number APL64 uses subsequently is known as a tie number. The particular function you use determines how the system interprets the identifier argument, the way the file is shared, and the expected format of the file.

During the second stage of file use, the program accesses and modifies the content of the file, possibly repeatedly. During this stage, the program uses the tie number to refer to the connection. These operations use system functions such as `⎕fread` or `⎕nread` to read data from a file, `⎕fappend`, `⎕fappend` or `⎕nappend` to append data at the end of the file, and `⎕freplace` or `⎕nreplace` to update data within the file. You may use other functions to inquire about the state of a tied file, modify its access matrix, and rename, erase, or reorganize it.

The third stage of file use is to break the connection between the program and the file. This is accomplished by the functions `⎕funtie` and `⎕nuntie`. These functions take one or several tie numbers as their argument.

Identifying a File

To begin using a file in APL64, you identify the file and assign a tie number. There are several functions you can use to assign a tie number to an existing file. You can also create a new (empty) file and and assign a tie number to it. Thereafter, the tie number usually is sufficient to allow you to use the file. Some file functions require an identification in addition to an established tie number. If you want to rename or duplicate a component file, you must already have tied the file and you must identify the target name. (You can duplicate a component file to the same name, which has the effect of compressing it.) If you want to erase a file from APL64, you must already have tied it and you must identify it for the operating system as well. (Naturally, you can rename, copy, and delete files outside of APL64 using the facilities of the operating system.)

Certain aspects of naming files can be surprisingly complex. In addition, the rules have changed over time, and they are still changing as operating systems evolve. The next sections of this chapter detail the considerations of identifying files in APL64. If you use simple, and short, file names, you can skip the first quarter of the chapter except for the syntax of tying files.

Background

The traditional handling of files in APL assumed DOS naming conventions, using only a limited character set for names, and restricting name length to eight characters, plus a three-character extension that followed a period (commonly called a "dot" when referring to file names). In some cases, the extension was mandatory for certain file types. Directory names in the path were similarly limited.

- File and directory names may be longer than eight characters.
- Extensions may be longer than three characters.
- Multiple "dots" are allowed in names.
- Names may contain spaces as significant characters.
- Many unusual characters are supported in names, including the alphabetic characters with codes above `128` and some special characters with codes below `128`.

Detailed rules and limits are defined, implemented, and enforced by the individual file system holding the data. Provisions exist for third party installable file systems; such systems may be local to a particular machine or accessed over a network. Although existing file systems are commonly insensitive to case, Windows uses calls that allow for case sensitive file systems and systems that do not record case at all.

Short Alias File Names

To facilitate compatibility with older software and operating systems, some file systems that support long file names create short alias names so that every file can be referred to by an `8.3` format name. Windows generates short names using an algorithm that removes spaces and some unusual characters, truncates the name sufficiently, then appends a tilde (~) and at least one digit. It chooses digits to ensure the short name is different from other short names in the same directory. In a particular directory, the files in the first column below could have the short names in the second column.

```
foo bar number 1.txt stuff         FOOBAR~3.TXT
foo bar number 2.txt               FOOBAR~2.TXT
```

Note that the association between the names is tenuous. If you duplicate the file to the same name, the relationship between the long name and the short name may be severed.

Long File Name Support

With the advent of Windows 95, APL2000 extended the set of system functions that deal with files to provide APL application developers nearly complete access to the Windows file name space. The Windows facilities that APL64 supports include: Universal Naming Convention (UNC) names, long file and directory names, special and extended ANSI characters in names, long path names, unobscured host operating system error reports, and wild card file searches, as well as file names that match the traditional `8.3` pattern. The system also allows you to choose file extensions for component files.

This extended functionality was implemented by creating a set of additional system functions to create, name, tie, and delete files. The names of the new functions parallel the traditional functions with an "X" inserted as the initial letter. Thus, the traditional functions `⎕fcreate` and `⎕ncreate` that you used to create the two types of files (with the traditional naming limitations) gained companion functions named `⎕xfcreate` and `⎕xncreate` that also create the two types of files, but use the expanded naming capabilities. The files thus created are no different than those created with the traditional functions, except possibly for the name. The system functions that manipulate the files within APL, such as those that write to a file after it is tied, work with files created by either the traditional or the extended functions.

APL2000 created the new functions while leaving the traditional functions intact so that there would be no impact on the existing code base. It was anticipated and expected that APL programmers would migrate to the extended functions over time, as new applications are written and the use of long file names becomes ubiquitous. APL2000 now recommends that you do so, because we cannot guarantee that future operating systems will continue to support DOS assumptions, even though the traditional APL file functions will continue to be supported. As an indication, current Microsoft operating systems allow short names that are longer than eight characters. If future operating systems no longer carry a short name equivalent for the full file name, the traditional APL file functions will have nothing to identify the file.

The extended file functions can be used with traditionally formatted names. If you want to update existing applications, you can convert the traditional file functions to the extended file functions by inserting the X in each function name with a few conditions that are identified below.

Traditional File Identifiers

A traditional component file name consists of from one to eight uppercase letters or numeric characters; it must begin with a letter. The system assigns APL component files the extension `.SF`. A traditional native file name can contain any characters permitted by Windows DOS naming conventions and is not restricted to beginning with a letter. Depending on your operating system, you may be able to use names that are longer than eight characters as traditional names, as long as they do not contain spaces or unusual characters. APL file functions allow the tilde character in directory names and in the names of existing files to accommodate short file names. You cannot, however, use a tilde in file names you create using the traditional functions.

You can always use a complete path name to identify a file; for example, `C:\APL\FILES\PERSONS`. The disk drive designation and directory path are optional. If you omit them, the system assumes the current directory. When you use a file name with or without the path identification as an argument to a file function, you represent that identifier as a character vector enclosed in quotes, or as an APL expression whose value is a character vector.

Traditional APL systems also provide a mechanism for associating a library number with a DOS directory. Libraries are much like directories except they are identified by numbers instead of names and they are not hierarchical. You use a positive integer for the library number. If you have a library defined, the library number and file name identify the file. You must separate the library number from the file name with at least one space; for example, `3 PERSONS`.

You can use libraries to allow applications that were developed on other APL+ systems to run without modification. However, note that libraries do not work with extended file functions. APL2000 recommends that you not use this obsolete functionality for new development.

Extended File Identifiers

Most of the extended file functions have a file identifier as an argument. An extended file identifier is a relative or absolute path name ending with the file's name and extension. Path names may be preceded, optionally, by a drive letter and colon. Absolute path names begin with a delimiter (`'/'` or `'\'` in Windows, denoting the root of the directory tree). Relative path names do not begin with a delimiter and are taken as relative to the current directory. Each of the following could be valid and meaningful file identifiers:

```
      '\foo/goo\hoo.dat'
      '.../foo/goo/hoo.dat'
      'c:\winnt\system\stuff\..\..\where ever'
```

The detailed rules for the interpretation of these strings are defined, implemented, and enforced by the underlying operating system and the attached file systems. These rules are not uniform among systems; for example, case may or may not matter and may or may not be preserved. There appear to be slight differences in the (unusual) characters allowed in file names by various systems.

The intent of the design for the extended functions is that an extended file identifier argument is a string of characters passed to the host operating system without interpretation or modification. The host uses the string as a file name, according to its rules, whatever they might be.

When an operating system call fails, perhaps because an argument was inappropriate, the APL interpreter includes the operating system's error information in `⎕dm`. Thus, two systems may return different error messages for the same sequence of actions; some messages may be misleading or obscure.

In many cases, an identifier argument, say for `⎕xntie`, will simply be an APL character vector, perhaps originating as quoted text in the APL program. For example, the statement

```
      'c:\winnt\system.ini' ⎕xntie ¯5
```

will have the obvious intended effect.

Similarly,

```
       '\\sigma\public\good stuff\latest data.sf' ⎕xftie 3
```

attempts to tie a component file on a machine named `sigma` in a directory with the sharename `public`. This is an example of the unversal naming convention for network shared files. The file's pathname contains directory names with embedded spaces and length not limited by the `8.3` format.

The extended functions neither require nor ensure that component files have an SF extension. The statement: `'mydata.sf' ⎕xftie 3` ties the same file as: `'mydata' ⎕ftie 3` but as an extended tie. The file identifier for the extended function must explicitly contain the `.sf` extension that is implicitly supplied by the traditional function.

You can use `⎕xfcreate` to create component files with any extension. You can tie them with `⎕xftie` and `⎕xfstie`. It is the programmer's responsibility to ensure the correct extension. Consider the statement:

```
       '\\sigma\public\good stuff\last data.xyz' ⎕xftie 3
```

The file itself has an xyz extension. Presumably, the programmer has good reason to believe the file's internal format is suitable for component file operations. If not, the `⎕xftie` or a subsequent file operation will fail, possibly mysteriously. Component file operations must be used only with uncorrupted APL component files. It is useful to give component files an extension of SF unless a different extension is really necessary; some system functions depend on that extension.

Character Sets and Encoding

The traditional tie functions allow only a restricted set of characters in file identifiers. The extended tie functions allow the use of any character acceptable to the host file system. (The design of the extended file functions will allow future extensions to support Unicode file names. Currently, only support for character codes less than `256` is actually implemented.)

An extended file function identifier in APL is a vector, the elements of which may be characters or integers. Mixed vectors of character and integer data are allowed. Conceptually, the character elements of this vector are replaced (in the interpreter) by their zero-origin position in `⎕av`; the resulting integers are treated as character codes by the host. Some examples of valid extended file names you can use are:

```
       'George Washington.sf' ⎕xfcreate 1
       (197 ,  'ngstrom.sf') ⎕xftie 2
       'ú.txt' ⎕xncreate ¯3
       ('G'  , 246 , 'del') ⎕xntie ¯4
```

The first line attempts to create an APL component file with the name `George Washington` and the extension `.sf`. The second line attempts to tie a component file with the name `Ångstrom.sf`. The third line attempts to create a native text file with extension `.txt`. See below for what the name might appear to be. The fourth line attempts to native tie a file with the name of `Gödel` (ANSI character `246` is a lowercase o umlaut). However, see below for some of the complications of using characters with values greater than `128`.

The characters in positions `32` to `126` of `⎕av` are the same as the characters in the corresponding positions of the extended ANSI character set used by Windows and the `16` bit Unicode set. For example, the character A is keystroke Alt+`065`. You could include this character in a file identifier argument as either `⎕av[65]` (that is, the letter A) or `65`.

The same is true for some characters outside the range specified above. For example, the character Å (uppercase A with circle above) is keystroke Alt+`0197`. You can use the Windows Character Map accessory with any Windows ANSI font to obtain the integer codes above `128`. In this case, the APL code for this character is also `197`, so entering Å between quotes in APL program text would also work as expected. You could include this character in a file identifier argument as either `⎕av[197]` or `197`.

Some other characters may work in the host operating system but not in APL. For example, in Windows 2000/XP you could create a filename consisting of the two characters © (Copyright) and ® (Registered) using the vector `169 174`. However, since these characters are not contained in `⎕av`, you cannot enter them as APL character data.

Many of the accented letters used by European languages are contained in both character sets; however, some have different positions. Using such characters between quotes in APL program text to specify an extended filename may not have the intended effect. For example, the lowercase u acute is `⎕av[163]` but ANSI `250`. If you specify

```
       'ú.txt' ⎕xncreate ¯3
```

the operating system may see the file with the English Pound sign (`£.txt`), depending on how you generate the character. (Note that in the APL session, in a GUI edit box, and in applications such as Windows Notepad, you can get different characters depending on whether you type Alt+`163` or Alt+`0163`.) If you specify the argument as an integer, `163 ⎕xncreate ¯3`, Windows Explorer sees the file with the English Pound sign. The character for `⎕av[250]` is the OR sign (∨).

Other characters are even more problematic. The lower case o umlaut is `⎕av[148]` but ANSI `246`. If you specify the o umlaut in APL, the host system will do something with its character `148`. In many fonts this character is the right half of paired double quote marks, but the character is not always available so the system may display an unspecified symbol.

Typically, application programs will acquire filenames via a GUI interface. By avoiding any translation (setting the translate property for the control to zero) between the ANSI and APL character sets, the system will correctly encode the data variable representing the filename for use with the extended file functions. You need translation only when the filename is entered or displayed by the APL session manager, function editor, or debugger. Such translation can be conveniently handled by a pair of utility functions such as `AV2ANSI` and `ANSI2AV` in the Windows.w3 workspace.

The names contained in `⎕xfnames` and `⎕xnnames` and those returned by `⎕xlib` are also represented in the Windows character set and may display incorrectly in an APL context. However, they are directly usable as *fileid* arguments to the extended functions. If you query and use the values under program control, the host system will recognize the file names; if you display them in your session, the characters may differ.

Using File Ties

To use a file, you must pair it with an integer file-tie number. The pairing of a file and a file-tie number is called a file tie; a file is said to be tied if such an association has been made. Most operations on the file refer to the file by its tie number rather than by its name. You tie APL component files with positive integer tie numbers and native files with negative integer tie numbers. The tie number can have any value with the appropriate sign that is different from any other file-tie numbers already in use. File ties exist only during an active session. When you close APL, the system unties the files automatically even if you have not explicitly untied them.

Tying Component Files

You can access an existing component file by tying it; the syntax of `⎕xftie` is:

```
'fileid.ext' ⎕xftie tieno
```

If you use passnumbers for security, the syntax of `⎕xftie` is:

```
'fileid.ext' ⎕xftie tieno passno
```

The file you name in the left argument of `⎕xftie` must already exist. You need not always use the same number to tie a specific file. A file-tie number used with `⎕xftie` can be any integer from `1` through the count limit allowed by the system (`2147483647`). File ties are "slippery" for a given user. That is, if you tie a file to one number, `⎕xftie` allows you to tie the file to the same number or to another unused tie number without requiring you to first untie the file.

A file remains tied until you untie it or end the APL session. Thereafter, you must re-establish a tie to use the same file. The status of tied files is not changed by any system commands except `)off`. Therefore, loading into or clearing the active workspace leaves active any ties you set previously during the session. This behavior means that you can load different workspaces without having to retie the files.

There are two basic ways for multiple users to access files: exclusively and concurrently. When users access a file exclusively, one user has access to the file at a time. No other user can work with the file until the current user is done. The functions `⎕xftie` and `⎕ftie` (note: no "x") allow exclusive access.

When users share a file concurrently, all the users can work with the file at the same time. The `⎕xfstie` function ties a file, as `⎕xftie` does. However, this shared tie permits others to share-tie the same file. You can use `⎕xfstie` to share-tie a file if the file allows any form of access, as long as no other user has the file exclusively tied using `⎕xftie`. The syntax is the same as for `⎕xftie`:

```
'fileid.ext' ⎕xfstie tieno
'fileid.ext' ⎕xfstie tieno passno
```

APL64 provides other system functions for processing files in a network environment where the files can be shared by multiple users concurrently. These functions allow you to control concurrent file sharing and limit other users' access to your files. See the "Sharing Files" section later in this chapter.

Traditional Functions that Create Component File Ties

The traditional functions for tying component files are very similar to the corresponding extended functions; however, the file identifier for traditional APL component file functions does not include an extension; the system assumes the extension (`.SF`) when it searches for an existing file and appends it when creating a file or changing its name.

The `⎕ftie` function attempts to establish a traditional component file tie between *tieno* and the file designated by *fileid*. The tie is exclusive.

```
'fileid' ⎕ftie tieno
'fileid' ⎕ftie tieno passno
```

The `⎕fstie` function attempts to establish a traditional component file tie between *tieno* and the file designated by *fileid*. The tie may be shared.

```
'fileid' ⎕fstie tieno
'fileid' ⎕fstie tieno passno
```

You can use either of the traditional functions to tie a file that has a longer name by specifying *fileid* as the traditional (`8.3`) representation that includes the tilde character (~).

Tying Native Files

You must use negative numbers as tie numbers for native files. Tie numbers for use with `⎕xntie` can be any integer from `¯1` through the negative of the count limit (`¯2147483648`). The `⎕xntie` function attempts to establish an extended native file tie between *tieno* and the file designated by *fileid*. The tie is shared. The optional *openmode* specifies open options.

```
'fileid.ext' ⎕xntie tieno
'fileid.ext' ⎕xntie tieno openmode
```

The syntax for the traditional function, `⎕ntie` , is the same, as the system does not have a default extension for a native file.

Mixing Traditional and Extended Ties

In addition to the two classes of file ties, component file ties and native file ties, the extended tie-creating file functions introduce a new, orthogonal classification of ties, traditional or extended, according to the function used to establish the tie. Thus, there are four classes of file ties: traditional component, extended component, traditional native, and extended native.

This classification of ties refers to the way a file is being used, not to the file itself. Both traditional and extended component file ties share the positive integers as tie numbers. Traditional and extended native file ties share the negative integers as tie numbers.

A file can be share-tied with both an extended tie and a traditional tie at the same time; however, the file sharing machinery sees the connections as two processes. Thus you cannot exclusively tie the file with either an extended or traditional tie while the other has it share-tied. That is, (in the same process) `⎕xftie` can replace a tie created by `⎕xfstie` or `⎕xftie` but not one created by `⎕ftie`, `⎕fstie`, `⎕ntie`, or `⎕xntie`.

Untying Files

The `⎕funtie` function breaks the association of an APL component file and tie number for both traditional and extended file ties. The argument is a vector of `0` or more file-tie numbers, so `⎕funtie` can untie several files at once. It has the following syntax:

```
⎕funtie tienos
```

The `⎕nuntie` function breaks the pairing of a native file and its tie number.

Creating and Building Files

Descriptions and examples of file operations in this chapter use the two files described in this section. If you create and execute the `SAMPLE1` function shown below, you will have the two files and can execute the later examples. Note that you will have to enter the numeric values for the `SALES` file in your session. The APL component file named `PERSONS` will have four components, each of which is a character vector. The APL component file `SALES` also will have four components, each of which is a numeric vector. The files should contain these values:

Component Number	**`PERSONS` File**	**`SALES` File**
1	`'SMITH'`	`5 6 3 1`
2	`'JONES'`	`2 6 1`
3	`'KELLEY'`	`4 6 2 9 1`
4	`'BECKER'`	`20 6 4`

Creating Component Files

The file function `⎕xfcreate` establishes a new APL component file; the function `⎕fappend` places new components in the file. The following program builds the two files `PERSONS` and `SALES`.

```
     ∇ SAMPLE1;NAME;PEOPLE;T;VALUES;CN
[1]    ⍝ Create the files 'PERSONS' and 'SALES'
```

```
[2]     'PERSONS.SF' ⎕xfcreate 5
[3]     'SALES.SF' ⎕xfcreate 20
[4]    PEOPLE←'SMITH' 'JONES' 'KELLEY' 'BECKER'
[5]   ⍝ Prompt for sales for each person
[6]   ⍝ Then store the values in the files
[7]   ⍝
[8]    :FOR NAME :IN PEOPLE
[9]         'Enter Values for ',NAME ⋄ VALUES←⎕
[10]        CN←VALUES ⎕fappend 20
[11]        'Values Stored in Component Number ',⍕CN
[12]       ⍝ Append to file 'PERSONS'
[13]        T←NAME ⎕fappend 5
[14]   :ENDFOR
[15]  ⍝
[16]   ⎕funtie 20 5
    ∇
```

Throughout the descriptions, *fileid* stands for an APL vector containing a file identifier; *tieno* is an integer tie number, positive for the component file functions, negative for native file functions; *passno* is an integer passnumber; *compno* is a component number; and *openmode* designates options for opening a native file.

The `⎕xfcreate` function creates a new file and prepares it for further operations. When `⎕xfcreate` creates the file, it associates the file name with a tie number. You use the tie number to perform subsequent file operations. The syntax is:

'*fileid*.*ext*' `⎕xfcreate` *tieno*

The file name must be different from that of any existing file in that directory, and the tie number must not be in use. When the system executes `SAMPLE1`, line `[2]` creates the file `PERSONS` and ties it to `5`.

Files you create the way `PERSONS` and `SALES` were created have no maximum size. You can specify a size limit for the file by specifying the number of bytes after the file identification in the left argument of `⎕fcreate`. Each time you attempt to add or replace material in the file, the system automatically checks whether the operation will cause the file to exceed the limit. You receive the error `FILE FULL` if you attempt to put more than that number of bytes of data into the file. The use of file size limits allows you to budget your disk storage and prevent runaway programs from filling the entire disk.

If line `[2]` of `SAMPLE1` had been

```
'PERSONS.SF' 10000 ⎕xfcreate 5
```

the size limit would be `10,000` bytes. The default value `0` indicates no size limit. You can also add, as the third element of the left argument, an integer specifying the number of the first component.

An empty vector may be used as a place holder for *size* or *compno* when the default values are intended. Note that when *size* or *compno* is specified, *fileid* must be nested within the left argument. The following are valid statements:

```
('G' 246 'd'  'e'  'l' ) ⎕xfcreate 10
'foo.sf' ⍬ 19 ⎕xfcreate 11
'goo.sf' 10000 17 ⎕xfcreate 12
```

Note, however, that the statement: `'g' 120 50 ⎕xfcreate 13` would create a component file with the name `gx2` rather than a file named `g` of size `120` starting at component `50`. To accomplish the latter, you would have to nest the first character with a statement like: `(,'g') 120 50 ⎕xfcreate 13` or the equivalent.

Creating and Building Native Files

You create native files in an analogous fashion. You use the `⎕xncreate` function to create a native file. Native files follow somewhat different naming conventions, and you use negative tie numbers with them. See `⎕ncreate` in the System Functions, Variables, and Constants chapter in this manual for more information.

```
'fileid.ext' ⎕xncreate tieno
```

Traditional Functions that Create Files

```
'fileid'                ⎕fcreate tieno
'fileid size'           ⎕fcreate tieno
'fileid size/compno'    ⎕fcreate tieno
```

The `⎕fcreate` function attempts to create a new component file and tie it to *tieno*. The name of the new file is specified by *fileid*. The tie is shared. Note the differences in syntax. You do not specify a file extension; the system appends it to the *fileid* you provide. If you specify the optional arguments, they are part of a single character vector, rather than being separate elements of a nested vector. The optional *size* is an integer scalar file size limit for the new file. The optional *compno* is an integer that specifies the starting component number for the new file. The slash is required if you specify the component number; if you omit size and specify the component number, you must include a space between the *fileid* and the slash.

The two statements below are the same except that the `⎕xfcreate` forms an extended tie.

```
'goo.sf' 10000 17 ⎕xfcreate 12
'goo 10000 /17' ⎕fcreate 12
```

The `⎕ncreate` function attempts to create a new native file and tie it to *tieno*.

Adding Data to Files

At the point you created it, the file `PERSONS` exists, but it is empty since it contains no components. Components are added on line `[13]` of the sample function. The `⎕fappend` function puts a new component, containing an APL value from the active workspace, at the end of an APL component file. The value in the workspace is not altered. The syntax is:

```
compno ← value ⎕fappend tieno
```

The left argument is the value for the new component. It can be the name of a variable or the result of any APL calculation. For example:

```
NAME ⎕fappend 5
(5+2×4 6⍴⍳24) ⎕fappend 5
```

The right argument is the tie number of the file that will contain the new data. The `⎕fappend` function returns the new component number as its result. Examples of appending to files `SALES` and `PERSONS` appear in lines `[10]` and `[13]` of `SAMPLE1`.

The `⎕nappend` function appends the contents of an array, byte for byte, at the end of a native file. However, there is no structure analogous to components in a native file. It is one array. There is no size checking in writing to a native file.

Reading Data from a Component File

The `⎕fread` function reads the value of a file component into the active workspace.

```
    ∇ SAMPLE2;I;X
[1]  ⍝ Total sales for each person
[2]   'PERSONS.SF' ⎕xftie 1
[3]   'SALES.SF' ⎕xftie 2
[4]   'NAME      SUM'
[5]   '----      ---'
[6]  ⍝
[7]  :for I :in ⍳4
```

```
[8]       X←⎕fread 2,I
[9]       (10↑⎕fread 1,I),⍕+/X
[10] :endfor
[11] ⍝
[12]  'COMPLETE.'
    ∇
```

You can place the value in a variable (as in line `[8]` of `SAMPLE2`), use it in an expression (as in line `[9]`), or display it directly at the terminal. The syntax of `⎕fread` is:

result `←` `⎕fread` *tieno compno*

The argument is a two-element vector. The first element is the tie number; the second element is the component number. The `⎕fread` function returns the value of the specified file component as its explicit result. Note that the function `SAMPLE2` tied the files `PERSONS` and `SALES` but did not untie them, so those two files remain tied as files `1` and `2`.

Reading Data from a Native File

Reading from a native file with the `⎕nread` function is more complicated, since the file is not logically divided into components and does not internally track the datatype of what has been stored there. You can read part of a native file if you know the length and starting byte of the data you want to read. See `⎕nread` in the System Functions, Variables, and Constants chapter in this manual for more information.

Replacing and Dropping Components

You can replace a component in an APL component file with any APL array using `⎕freplace`. You need not match the physical size in bytes, the datatype, or the shape of the array you are replacing. The syntax is

value `⎕freplace` *tieno compno*

In a file with `20` components, you could replace an integer in component `7` by a table of numbers without knowing how much room on the file the original component occupied:

```
ASTRO ⎕freplace 5 7
```

A common use of `⎕freplace` is to update a component by catenating a new value to it. In the following example, the value of the variable `DEB` is catenated to the existing value of component `3` of the file tied to `7`:

```
((⎕fread 7 3),DEB) ⎕freplace 7 3
```

Note: In this example, the new component is larger than the one it replaces. The new component may be written at the physical end of the file, which leaves the previous size of the component as wasted space. This is known as an exploding `⎕freplace`. The amount of wasted space in a file is reflected in the last element of `⎕fsize`. You can reclaim this space using `⎕fdup`. For more information, see "Compacting a Component File" later in this chapter.

The `⎕fdrop` function removes components from either end of an APL component file. The syntax is

`⎕fdrop` *tieno n*

The argument to `⎕fdrop` is a two-element vector. The first element is the tie number of the file. The second element is an integer that specifies both the number of components you want to drop, and from which end of the file you want to drop them. **Note**: If you drop any components, you cannot recover them. Be sure you drop them from the correct end of your file.

- If *n* is positive, `⎕fdrop` removes the components from the front of the file.
- If *n* is negative, `⎕fdrop` removes the components from the end of the file.
- If *n* is `0`, `⎕fdrop` removes no components.

The component numbers of the remaining components do not change. For example, if the file tied to `99` has `10` components numbered `1` through `10`, after running

```
        ⎕fdrop 99 4
```

the file has six components remaining, numbered `5`, `6`, `7`, `8`, `9`, and `10`. If you subsequently run

```
        ⎕fdrop 99 ¯2
```

the file has components `5`, `6`, `7`, and `8`. You cannot drop components from the interior of a file. You can use other techniques to signify that an interior component holds no information and should be bypassed in later processing. One way is to replace the component with an empty vector (`''`).

Replacing Data in a Native File

The native file operation to replace data, `⎕nreplace`, can only replace data byte-for-byte. You (or the program) must supply the position in the file where you want replacement to begin. The new value replaces exactly the amount of the file needed to store it, regardless of what was there before. If you try to store more data than the remaining space in the file can accommodate, `⎕nreplace` lengthens the file to make room for the rest of the data.

There is no native file operation that corresponds to `⎕fdrop`, since native files are not organized into components. You can, however, shorten a native file by resizing it to drop bytes off the end.

Erasing Files

When you no longer need an APL component file, you can erase it using `⎕xferase`; to erase a native file, use `⎕xnerase`. The syntaxes are:

'*fileid*.*ext*' `⎕xferase` *tieno*
'*fileid*.*ext*' `⎕xnerase` *tieno*

The `⎕xferase` and `⎕xnerase` functions delete files from the directory. The space they occupied is then made available for use by other files. Since the files no longer exist, the file ties are also broken. You must tie a file before you can erase it, and you must specify both the file name and the tie number exactly as you specified them when you tied the file.

Traditional Functions that Erase Files

'*fileid*' `⎕ferase` *tieno*
'*fileid*' `⎕ferase` *tieno passno*

tieno must be an exclusive traditional component tie. The system attemptes to erase the component file tied to *tieno*. The interpreter processes *fileid* exactly as if it were tying the file. Then it checks that the resulting complete path name matches the one produced at the time *tieno* was tied. If the path names are different, the function fails and the file is not erased. This is a perfunctory check intended to avoid immediate execution calamities . The interpreter does not attempt to recognize different names for the same file (UNC vs drive letters, longnames vs `8.3` aliases, etc.).

'*fileid*.*ext*' `⎕xnerase` *tieno*

tieno must be an extended native tie. The native file tied to *tieno* is erased. The interpreter checks *fileid* as described for `⎕xferase`.

Merging Values

The following function shows how you can change the arrangement of filed information. Suppose you want to merge the components of `PERSONS` and `SALES` into a new file called `RECORDS`. Each odd-numbered component will come from `PERSONS` and each even numbered component from `SALES`. The function erases the two original files after the merge is complete. Recall that `PERSONS` and `SALES` are still tied following execution of `SAMPLE2`.

```
     ∇ SAMPLE3;T;I
```

```
[1]    ⍝ Merge files tied to 1 and 2 into 'RECORDS'
[2]     'RECORDS.SF' ⎕xfcreate 3 ⋄ I←1
[3]    ⍝
[4]     :REPEAT
[5]        T←(⎕fread 1,I) ⎕fappend 3 ⍝ From 'PERSONS'
[6]        T←(⎕fread 2,I) ⎕fappend 3 ⍝ From 'SALES'
[7]     :UNTIL 4<I←I+1
[8]    ⍝
[9]     ⎕funtie 3
[10]    'PERSONS.SF' ⎕xferase 1 ⋄ 'SALES.SF' ⎕xferase 2
    ∇
```

Note that after the function merges the files, it unties the new file `RECORDS` (tied to `3`). It is not necessary to untie the files tied to `1` and `2` because erasing files unties them automatically.

Using File Management Facilities

Inquiring about File Ties

Corresponding to each of the five tie classes are tie inquiry functions which return the tie numbers and file identifications of the ties in the corresponding classes. These system functions are all niladic and may return an empty result. The following five functions return a vector of tie numbers for current ties of the appropriate type:

`⎕fnums`	Traditional component ties
`⎕xfnums`	Extended component ties
`⎕nnums`	Traditional native ties
`⎕xnnums`	Extended native ties

Note that `⎕xfnums` does not include the tie numbers assigned by the traditional component file functions, and `⎕fnums` does not include tie numbers assigned by the extended component file functions. Thus, if you get a file tie error, you must check both functions to determine if the file is already tied. However, for `⎕cfnums`, the tie numbers returned have no relationship to `⎕fnums` or `⎕xfnums`. This means that a colossal file and a traditional or extended file can have the same file tie number.

The following five functions return character matrices, each row of which is a file identifier for a file tie. The rows of these functions correspond in order and number to the elements of the corresponding function that returns tie numbers.

`⎕fnames`	Traditional component ties
`⎕xfnames`	Extended component ties
`⎕nnames`	Traditional native ties
`⎕xnnames`	Extended native ties

Unlike `⎕cfnames`, the format of `⎕fnames` allows for library number definitions and file names. Each row of `⎕xfnames` or `⎕xnnames` is a complete path name, including drive letter and colon or UNC prefix, exactly as passed to the host operating system to open the file at the time it was tied. The traditional functions use only uppercase, and the extended functions preserve case as it was entered or stored.

For a tabular display of component files tied with the extended functions, use:

```
      ⎕xfnames, ⎕xfnums
D:\APL\Aplwin60\SAMPLE.SF   1
D:\APL\Aplwin60\PERSONS.SF 2
```

You can use the corresponding expression with the functions ⎕fnames and ⎕fnums to display all the files tied with the traditional functions. The vector returned by ⎕xfnums is exactly what ⎕funtie needs to untie all the files tied with the extended functions. You can untie all the APL component files tied with both the extended functions and traditional functions with the statement:

```
        ⎕funtie ⎕xfnums , ⎕fnums
```

Inquiring about File Open Mode

File tie state information is available via the monadic variant of the ⎕ntie, ⎕xntie, ⎕xftie, ⎕ftie, and ⎕cftie functions. Monadic forms of these functions take a tie number as the right argument and return a two-element integer vector indicating the file tie state.

The values of the first element are the same as the open mode arguments to the dyadic version of ⎕ntie and indicates what the current open mode of the file is. These same values apply for the component ties. The open mode values are sums of the following:

Access Requested	**Access granted to other users**
0 = read access	16 = no access allowed (exclusive)
1 = write access	32 = read access allowed, write access denied
2 = read and write access	48 = write access allowed, read access denied
	64 = read and write access allowed

For native file ties, if an open mode argument was specified when the file was tied, the result of monadic ⎕ntie is that value. If no open mode argument was specified the return value is dependent on whether read or write access is granted and what value of <network> was specified in the APL configuration file. For the component and colossal component file ties, the values are dependent on whether the tie request was for a shared or exclusive tie and whether read or write access is granted.

The second element indicates if the tie was share tie (i.e. ⎕fstie, ⎕xfstie, or ⎕cfstie) or an exclusive tie (i.e. ⎕ftie, ⎕xftie, ⎕cftie). Share ties show a value of 0, exclusive ties show a value of 1. The values of the second element are applicable only to component file ties. The second element value is meaningless for native file ties.

Renaming a Component File

The ⎕xfrename function changes an APL component file name. The syntax is

```
        'fileid.ext' ⎕xfrename tieno
        'fileid.ext' ⎕xfrename tieno passno
```

tieno must be an exclusive extended component tie. The ⎕xfrename function does not create a second copy of a file. After executing the following example, REPORTS no longer exists.

```
        'REPORTS.SF' ⎕xftie 100
        'OLDRPTS.SF' ⎕xfrename 100
        ⎕funtie 100
```

In addition to changing the name, ⎕frename also sets the ownership of the file to match the user number of the person who performed the operation. You can also use ⎕xfrename to move a file from one directory to another on the same drive.

The ⎕xfrename function fails if the new *fileid* is already in use. Under some unusual conditions, involving races between multiple concurrent processes, this function may complete without having finished its work or fail leaving the tie broken.

Traditional Function that Renames Component Files

```
        'fileid'     ⎕frename tieno
```

```
'fileid size' ⎕frename tieno passno
```

The component file tied to *tieno* is renamed to *fileid*. The tie must be exclusive. If you specify the optional size parameter in the left argument, you can limit the size of the file as you can when you create a file. Note that you cannot rename a file and change its size with the extended function. You can change the size of any component file using `⎕fresize`.

Copying a Component File

The `⎕xfdup` function makes a copy of the contents of an APL component file in another APL component file; however, despite the function's name, it is not an exact duplicate. Wasted space, such as from dropped or shortened components, is compressed out of the file. The components are physically reordered to match their numeric order, starting with `1`, by default, or whatever number you specify in the argument. If you want to maintain a file size limitation you must respecify it; any size or numbering specifications on the original file are not carried over. The syntax is the same as for creating a file.

```
'fileid.ext'                  ⎕xfdup tieno
'fileid.ext' size             ⎕xfdup tieno
'fileid.ext' size compno      ⎕xfdup tieno
```

The `⎕xfdup` function attempts to copy a component file tied to *tieno*. The name of the new file is specified by *fileid*. The same considerations of syntax apply to `⎕xfdup` as apply to `⎕xfcreate`. The copy need not be in the same library or directory, or even on the same drive or machine, as the file you are copying, and the name need not be the same.

```
'FIGHT.SF' ⎕xftie 48
'OREGON.CMP' 10000 101 ⎕xfdup 48
⎕funtie 48
```

The `⎕xfdup` function does not change the ownership of the original file, but the user number that used `⎕xfdup` is the owner of the copy. You need not untie the copy of the file because it was never tied. If you tie the copy and compare its size to the size of the original file, you may find that the new file is smaller. The `⎕xfdup` function eliminates any wasted space while copying (see "Compacting a Component File," below).

You can also use the DOS `COPY` command, either from DOS or with `)cmd`, or under program control with `⎕cmd`, to copy component files without compacting them. In fact, because `⎕fdup` always compacts a file, you may not want to use it for routine file copies. You can use the DOS `COPY` command instead.

Copying a Component File with the Traditional Function

```
'fileid size/compno' ⎕fdup tieno
```

The `⎕fdup` function attempts to copy a component file tied to *tieno*. The name of the new file is specified by *fileid*. The same considerations of syntax apply to `⎕fdup` as apply to `⎕fcreate`.

Determining the Size of a Component File

The `⎕fsize` function displays size information about a component file. Its syntax is

```
result ← ⎕fsize tieno
```

The result is a five-element vector. The first two elements are the component limits of the file: the number of the first component, and a number that is one higher than the number of the last component. The component limits of a newly created file are `1 1`.

You can use the following expression in a program to establish the number of components in a file:

```
|-/2↑⎕fsize tieno
```

The third and fourth elements of the result are the amount of file storage currently occupied by the file and the file storage limit, in bytes. If the fourth element is `0`, the file has no imposed size limit other than available space on the disk. The fifth element of the result is the number of bytes of allocated but wasted file storage occupied by the file. You can reclaim this space using the `⎕fdup` or `⎕xfdup` functions, depending on how the file was tied. Suppose the result of `⎕fsize` for the file tied to `10` is:

```
      ⎕fsize 10
1 126 259584 265000 120
```

The result of `⎕fsize` indicates that the components in the file are numbered `1` through `125`, that the size limit for the file is `265,000` bytes, of which a total of `259,584` bytes are now occupied, and that there are `120` bytes of wasted storage space in the file.

You can use these functions not just to monitor the size of your files, but also in processing your data. For example, you can collect data whenever information is available and process it when the file becomes large and unwieldy, or at some fixed interval; for example, once a day or once a week. The following examples show how to implement two common file organizations — first-in, first-out (FIFO) and last-in, first-out (LIFO) — using `⎕fdrop` and `⎕fsize`. The examples use a file tied to `50`. You accumulate your data using a statement such as:

```
      data ⎕fappend 50
```

Then you can process the information with the following statements:

```
FIFO Organization             LIFO Organization
SIZE ← ⎕fsize 50              SIZE ← ⎕fsize 50
INFO ← ⎕fread 50,SIZE[1]      INFO ← ⎕fread 50,SIZE[2]-1
(process info)                (process info)
⎕fdrop 50 1                   ⎕fdrop 50 ¯1
```

The next example shows a function that uses `⎕fdrop` with a file whose components are released in an arbitrary sequence. `SAMPLE4` produces an invoice from information in a file named `CDATA`. After producing the invoice, `SAMPLE4` drops the appropriate component of `CDATA` (if it is the first component) or replaces it with an empty component (if it is not the first component). Each time you run `SAMPLE4`, the function checks to see if the last component of data is an empty vector. If it is, `SAMPLE4` drops that component. This technique tends to minimize the file size.

In line `[4]` of `SAMPLE4`, the function branches to the label `ERR:` if `N` is not within the range of the lowest and highest component numbers. Otherwise, line `[6]` reads the component, branching to `ERR:` if the component is empty. Line `[7]` calls the function that produces an invoice based on the record. Lines `[12]` through `[19]` drop empty components, if any, from the end of file.

This example shows one possible file organization technique. There are more static forms of database file organization that subdivide the data and keep directories. You may also want to use the convention of reserving the first component for a file description.

```
    ∇ SAMPLE4;LIM;N
[1]  'CDATA.SF' ⎕xftie 5
[2]  'Enter Record Number' ⋄ N←⎕
[3]                      ⍝ Test component number
[4]  LIM←2↑⎕fsize 5 ⋄ →((N<LIM[1])∨N≥LIM[2])/ERR
[5]                      ⍝ Read record and produce invoice
[6]  RECORD←⎕fread 5,N ⋄ →(0=×/⍴RECORD)/ERR
[7]  PRINTINVOICE RECORD
[8]            ⍝ Drop any empty components from front of file
[9]  :if LIM[1]=N  ⋄  ⎕fdrop 5 1  ⋄  LIM←2↑⎕fsize 5
[10] :else         ⋄  '' ⎕freplace 5,N
[11] :endif
[12]                     ⍝Test for empty file
```

```
[13] →(LIM[1]=LIM[2])/END
[14]                      ⍝ Test for empty component
[15] :while (0=×/⍴⎕fread 5,LIM[2]-1)
[16]                      ⍝ Drop empty component
[17]        ⎕fdrop 5 ¯1 ⋄ LIM[2]←LIM[2]-1
[18] :endwhile
[19] →END
[20] ERR:  'RECORD NUMBER NOT IN FILE'
[21] END:  ⎕funtie 5
    ∇
```

Changing a Component File's Size Limit

The `⎕fresize` function changes the size limit on an APL component file. You can use `⎕fresize` to impose a size limit on file, to increase a size limit, or to decrease a size limit. The syntax is

newsize `⎕fresize` *tieno*

where *newsize* is the new file size limit. Sometimes a file needs to hold more data than its current size limit allows. When a file does not have enough room to store the value, a `FILE FULL` error occurs:

```
      'OLDRPTS.SF' ⎕xftie 12
      REPORT ⎕fappend 12
FILE FULL
      REPORT ⎕fappend 12
      ^      ^
```

The fourth element of the result of `⎕fsize` shows the size limit for the file `OLDRPTS` is `100,000`:

```
      ⎕fsize 12
1 34 99512 100000 120
```

The `⎕fresize` function can increase the file size limit so that there will be enough room to append the new data:

```
      200000 ⎕fresize 12
      ⎕fsize 12
1 34 99512 200000 120
      REPORT ⎕fappend 12
34
```

You can also decrease the size limit of a file, provided you do not specify less storage than the data in the file already use. In addition, you can remove the size limit restriction completely:

```
      0 ⎕fresize 12
      ⎕fsize 12
1 34 99512 0 120
```

Compacting a Component File

Since storage space is limited, you may need to reclaim file space that contained data discarded by the use of `⎕fdrop` or `⎕freplace`. APL64 does not automatically release all the space that was occupied by discarded data within APL component files. If the system drops a component from the physical end of the file, it releases the storage; otherwise, you must use `⎕fdup` or `⎕xfdup` to reclaim the storage. Note that the components are not necessarily arranged on the disk in order, so the last physical component may not be same as the last numbered component. The best approach is to monitor `⎕fsize[5]` and reclaim wasted storage when that number becomes too large. You can reclaim storage space by using `⎕fdup`; for example:

```
      'FIGHT.SF' ⎕xftie 1844
      'FIGHT.SF' ⎕xfdup 1844
      ⎕funtie 1844
```

When you use `⎕xfdup` (`⎕fdup`) to compact a file, you must tie the file exclusively; that is, use `⎕xftie` (`⎕ftie`). The user number that uses `⎕xfdup` (`⎕fdup`) to compact the file becomes the owner of the file. Note that when you specify the same file name for the file you are duplicating and the resulting copy, there is only one file after you run `⎕xfdup` (`⎕fdup`). If you specify a different name as the argument to `⎕xfdup` (`⎕fdup`), you have both the original file and the compacted one.

If you know you are going to be adding data to a component, you can ameliorate the problem of exploding `⎕freplace` by creating the component with a large amount of dummy data; then when you initialize the component with real data, each addition will fit into the original space.

Getting Information about Files and Components

The `⎕fhist` (history) function provides information about an APL component file. Its syntax is

`⎕fhist` *tieno*

The function returns a three-row matrix with the following information.

- Row `1`: the user number of the file owner and the timestamp of the file's creation in both packed form and `⎕ts` form.
- Row `2`: the user number and timestamp associated with the most recent change to the file.
- Row `3`: the user number and timestamp associated with the most recent setting of the file access matrix.

The `⎕frdci` function returns component information. Its syntax is:

result `←` `⎕frdci` *tieno compno*

The result of `⎕frdci` is a ten-element numeric vector holding the following information.

- The workspace storage needed to hold the component's value.
- The user number of the person who last replaced or appended the component.
- The time that the component was last replaced or appended, in a packed-timestamp form given in microseconds since `00:00`, `1` January `1900`.
- The component timestamp in a seven-element unpacked form (year, month, date, hour, minute, second, millisecond).

`⎕frdci` is particularly useful in data collection or audit trail applications, where several users may have access to add data to a file. As each user adds data, the the system automatically tags the component with the time and the user number. Later, when a user with `⎕fread` and `⎕frdci` authorization processes the file, the source of each component is clear.

Listing Files in a Directory

The `⎕flib` function lists the names of files with the `.SF` extension in a specific directory. The syntax is

result `←` `⎕flib` *lib*

where *lib* stands for the directory name (or library number, in the style of earlier systems). The *result* is a character matrix. Each row holds the identification of a file in the designated directory. All of the APL component files (with the default extension) in a directory appear in the result of `⎕flib`, even if the user number that requests the display is not authorized to use the files. Note that these names are long names from the operating system unless you have set `SHORTNAMES=1` in the [Config] section of `APLW.INI`.

You can list the APL component files in the current default working directory by using an empty vector as the argument. The system command `)flib` yields the same information as `⎕flib`.

`⎕lib` and `)lib` return a list of all files (native files, APL component files, and workspaces) in a format consistent with native file names.

The Extended Library Function

`⎕xlib` *directory*
`⎕xlib` *pattern*

The `⎕xlib` function returns a matrix of the long filenames of all files in the directory you specify or that satisfy a pattern you specify. Note that the extended function preserves the case of filenames as the system records them, whereas the standard functions always use uppercase. Since case matters in sorting, `⎕xlib` may return the same list of names as `⎕lib`, but in a different order.

directory is a vector, possibly heterogeneous, formed like a *fileid*. It names a directory to be searched.

pattern is like *fileid* but may contain the wild-card characters `⋆` and `?` in the final component. The question mark represents exactly one character; it can be any valid character. The asterisk represents any number of characters, from zero to the maximum allowable length. You can specify multiple `⋆` and `?` characters. Only files matching this pattern are included in the result. If you specify a period in the pattern, the period must exist in the internal representation of the file. Thus a designation of `⋆.⋆` will not match a file that has no extension. The following examples are valid uses of the extended library function.

```
⎕xlib '⋆.sf'
⎕xlib '\\APLW\6.0.10\test files\⋆.sf'
⎕xlib '⋆FOO⋆.xyz'
⎕xlib '??xyz.⋆'
```

Managing Native Files

APL64 provides system functions and system commands for some of the file management tasks for native files that correspond to the component file functions described above.

- `⎕lib` and `)lib` return a list of all files (native files, APL component files, and workspaces) in a format consistent with native file names.
- `⎕nnames` and `⎕nnums` report on the status of traditional native file ties.
- `⎕xnnames` and `⎕xnnums` report on the status of extended native file ties.
- `⎕nrename` renames a native file tied with a traditional tie or moves a file into a different directory.
- `⎕xnrename` renames a native file tied with an extended native tie or moves it into a different directory.
- `⎕nsize` returns a single number representing the number of bytes in use. Since native files have neither components nor automatic checking for maximum size, there can be no meaningful equivalents for the other numbers in `⎕fsize`.
- `⎕nresize` alters the size of a native file on disk. You can use this function to pre-allocate space to reduce fragmentation and to discard material at the end of a file.

Not all the component file functions have native file counterparts. For example, there is no system function for duplicating native files. You can use the DOS COPY command, either from DOS or with `)cmd`, or you can use `⎕cmd` under program control to accomplish this.

There is no system function for compacting native files, since the system does not track their contents, and there is no system function that tracks the history of a native file.

Error Reports for Extended File Functions

For errors directly associated with the failure of a host operating system call, the extended file functions give an `XFHOST ERROR`. This error report contains all available information about the immediate circumstances of the failure. An error report might look like the following.

```
XFHOST ERROR _sopen 1005 2 2
The system cannot find the file specified.
'Foo Barr Esquire' ⎕xftie 9
                   ^
```

This report can be interpreted as:

`XFHOST ERROR` -- a fixed name for this kind of error

`_sopen` -- the particular Windows system or library call that failed. Note that this call may be part of a sequence of calls all of which must succeed to complete the APL function.

`1005` -- a designator for a particular place in the APL interpreter.
These numbers are subject to change with each build of the interpreter and are meaningful only with a complete system version number.

`2` -- the C library errno number, which is derived from `GetLastError()`.

`2` -- the number returned by the Windows `GetLastError()` call. Microsoft does not enumerate the numbers that can be returned for each system call. See the Microsoft file `winerror.h` for a list of possibilities.

The system cannot find the specified file; Windows supplies this text for error `2`

- `⎕tcnl` delimits the error report from the failing statement
- `'Foo Barr Esquire' ⎕xftie 9` is the failing APL statement
- `⎕tcnl` delimits the failing statement from the caret
- `^` (a caret) points to the failing APL function

Sharing Files

Sharing files typically implies multiple users on a network. The capabilities described in this section work within APL64, but are dependent on network resources. When users access a file exclusively, one user has access to the file at a time. No other user can work with the file until the current user is done. The functions `⎕xftie` (or `⎕ftie`) and `⎕funtie`, described earlier, allow exclusive access.

When users share a file concurrently, multiple users can work with the file at the same time. This section describes the mechanisms you can use to control concurrent file sharing and limit other users' access to your files. Two file system functions allow concurrent file sharing: `⎕xfstie` (or `⎕fstie`) and `⎕fhold`.

The `⎕xfstie` function ties a file, as `⎕xftie` does. However, this shared tie permits others to share-tie the same file. You can use `⎕xfstie` to share-tie a file if the file allows any form of access, as long as no other user has the file exclusively tied using `⎕xftie` (or `⎕ftie`). The syntax is:

`'`*fileid*`' ⎕xfstie` *tieno*

When several users use a file concurrently, their file operations proceed asynchronously. One user's processing may be interleaved with another's. For example, suppose that users P and Q have a file tied to `77` and are trying to add `1` to the value of component `5`. They use the following statement:

```
(1+⎕fread 77 5) ⎕freplace 77 5
```

When both users finish, component `5` should be increased by `2`. However, if the parts of the statement are executed in the following sequence, the value of component `5` is increased by only `1`, since P's program read the component before Q stored a new value.

User P	**User Q**
	`⎕fread 77 5`
`⎕fread 77 5`	
	(add `1`)
	`⎕freplace 77 5`
(add `1`)	
`⎕freplace 77 5`	

An interlock can prevent P's program from executing any part of an APL statement while Q's program is executing it (and vice versa). The `⎕fhold` function provides this interlock; its syntax is:

`⎕fhold` *tienos*

The `⎕fhold` function places an interlock on each file whose tie number is included in the right argument. The concept of this interlock is subtle; in effect, using `⎕fhold` means "wait until no one else has these files held, then hold them for me."

As executed by P, `⎕fhold` has three effects.

- Any interlocks in effect from a previous `⎕fhold` executed by P are released (even if they held the same file).
- P is placed in a queue behind every other user who has already executed `⎕fhold` for any of the files specified by P. P's program is delayed until no other user holds any of these files.
- Interlocks are set simultaneously on all of the designated files, and P's program then resumes execution.

`⎕fhold` provides a means for two or more users to cooperate and avoid conflict in file use. For example, if the program executed by P and Q in the previous example is changed to:

```
⎕fhold 77
(1+⎕fread 77 5) ⎕freplace 77 5
⎕fhold ⍳0
```

The first user to execute `⎕fhold 77` can proceed without delay, while the other user's program is delayed until the first user's program releases the hold (in the example, by executing `⎕fhold ⍳0`). The sequence looks like this:

User P	**User Q**
	`⎕fhold 77`
`⎕fhold 77`	
(delay)	(proceed)
	`⎕fread 77 5`
	(add `1`)
	`⎕freplace 77`
	`⎕fhold ⍳0`
(proceed)	(proceed)
`⎕fread 77 5`	
(add `1`)	
`⎕freplace 77 5`	
`⎕fhold ⍳0`	
(proceed)	

When Q executes `⎕fhold 77`, the hold on the file prevents P from establishing a hold. In P's program, `⎕fhold` simply waits to hold the file, which happens when Q executes `⎕fhold ⍳0`. P can then hold the file and execution proceeds. Cooperating users can therefore avoid conflict. However, if Q does not use `⎕fhold` properly, then P cannot prevent conflict. Suppose P **is** using `⎕fhold`, but Q is **not** using `⎕fhold`. The interaction is as if P were not using `⎕fhold` at all:

User P	**User Q**
`⎕fhold 77`	
(proceed)	`⎕fread 77 5`
`⎕fread 77 5`	(add `1`)
(add `1`)	`⎕freplace 77 5`
`⎕freplace 77 5`	
`⎕fhold ⍳0`	

File holds do not prevent others from using the file while it is held. A file hold only delays the execution of `⎕fhold` in other users' programs. In other words, no two users can have the same file held at the same time. File holds do not block other file operations such as `⎕fread` or `⎕freplace`.

Because users must cooperate for `⎕fhold` to work, you may want to use a specific function to access a file. You can use a file passnumber (see the next section) to enforce the use of a given protocol.

All interlocks are released when the user who set them executes another `⎕fhold`, signs off, or enters immediate execution mode. Untying or retying a file releases any interlock set on it. `⎕fhold ⍳0` releases all interlocks.

The immediate execution case is particularly important to remember. If you type the following three statements as three different immediate execution inputs, the `⎕fhold` has no effect at all.

```
      ⎕fhold 1
      PROCESS 1
      ⎕fhold ⍳0
```

However, a compound statement works correctly, since there is no immediate execution input between `⎕fhold` and `PROCESS`:

```
      ⎕fhold 1 ⋄ PROCESS 1 ⋄ ⎕fhold ⍳0
```

The following examples illustrate the use of file holds.

- Several users are concurrently appending to a file but make no other use of the file. File holds are not needed since `⎕fappend` tracks components added to the file by number. Although you cannot predict the order of the various values, any request for a file operation always waits until a previous operation on the same file is complete. The result of `⎕fappend` lets each user know which component contains his or her value.
- Several users are reading and replacing components of a file. No two users ever reference the same component. For example, P's program refers only to component `1`, Q's program only to component `2`, and so on. No hold is necessary, since no conflict can occur on the concurrent use of a single component.
- An application involves the use of three files in which like-numbered components contain related data. One program updates the files while the other programs read the files concurrently. To ensure that no program reading from the files encounters a mixture of old and new data, the updating program has this appearance:

```
      ⎕fhold 21 22 23
  (update code)
  ⎕fhold ⍳0
```

The file-reading program needs file holds even though the program itself is not modifying the contents of any files. The file reading programs that are operating concurrently with the updating program shown above have this appearance:

```
⎕fhold 21 22 23
A ← ⎕fread 21,N
B ← ⎕fread 22,N
C ← ⎕fread 23,N
⎕fhold ⍳0
```

As these examples show, the need for file holds depends upon the interrelation of program and file structure. The design of any application involving concurrent use of files requires careful analysis for possible conditions of conflict between programs. You can resolve such conflicts with appropriate use of `⎕fhold`.

Controlling Access to Files

Owning a file means that you originally created it or most recently renamed it. Typically, you have access to every file that you own, and can give access to others. In APL64, a user is an individual. In a cooperative environment, each user can have a different user number, by which APL64 identifies individual users for file-sharing purposes. You can discover your user number by entering `1↑⎕ai`. The default value is `1`.

APL64 tracks the user number that created or last renamed the file (its owner). Through the use of the access matrix, the file owner and other authorized users can extend or limit the types of file operations that a given user number can perform. You can use nonzero passnumbers in the access matrix to limit operations.

Access to native DOS files is controlled by a less specific mechanism. You can use DOS open modes and byte range locks to control access for native files. You can also use the DOS ATTRIB command to control the type of access permitted to a file.

Understanding the File Access Matrix

Associated with every APL component file is an access matrix that records the users who are authorized to use the file and the APL component file operations that each can perform. The access matrix affects only the actions taken by an APL user with APL component file system functions. It offers no protection against use by non-APL programs or from APL programs that have tied the file as a native file. Every access matrix is an integer-valued matrix with three columns and any number of rows.

- Column `1` is the user number to which you want to give access; a zero gives access to all users.
- Column `2` indicates the operations a user can perform.
- Column `3` is a passnumber to the file.

For example, suppose you want to set up a file that allows all users only `⎕fread` access. You want to be able to test the application that accesses the file and obtain the same permission as another user. You would set up an access matrix like this:

`0`	`1`	`0`	`⎕fread`-only access to all but owner.
owner	`1`	`0`	`⎕fread`-only access to owner.
owner	`¯1`	`1234`	full access with passnumber to owner to perform other file operations.

The following table shows typical entries in an access matrix. The details of columns 2 and 3 are of interest when exercising detailed control over access; they are discussed later in this section.

Typical Access Matrix Entries

User Number	Encoding of Permitted Operations	Passnumber	Description
12	¯1	0	Access granted to user 12.
23	9	5197	Access granted to user 23.
0	1	0	Access granted to all other users.

Access matrices follow these rules.

- The user numbers in Column 1 can be those of any users. A 0 value in Column 1 refers to all users other than the owner or those specified elsewhere in Column 1.

  The owner has full access if that user number does not explicitly appear in Column 1 of the access matrix. If the owner's user number does appear in the access matrix, then access is granted according to the access matrix.
- The value in Column 2 indicates which operations the user is authorized to perform. Each operation subject to access control is associated with a value, called the access code, that is a power of 2. The sum of these access codes is the nominal value in Column 2.

  Formally, the value in Column 2 is the integer representation of a Boolean mask that has a bit for each controllable operation. If

  ```
      MASK ←→ (32⍴2)⊤VALUE
  ```

  then MASK[32-N] (origin 1) is the bit regulating the operation whose access code is 2⋆N. If the bit is 1, the user is authorized for the operation. Some bit positions are not associated with any operations; the value of these bits is immaterial. Thus, ¯1 grants authorization for every operation, since (32⍴2)⊤¯1 ←→ 32⍴1.

  This last property is often used to grant all but certain kinds of access to a file. The technique is to subtract from ¯1 the access codes for the operations to be denied. For example, ¯5 grants all but ⎕ferase access (¯1-4); ¯7 grants all but ⎕ferase and ⎕ftie access (¯1-(4+2)).
- The passnumber in Column 3 can be any positive or negative integer value. Together with the user number in Column 1, the passnumber determines which row of the access matrix is applied in verifying authorization for each operation. A single user or class of users can have multiple rows in the access matrix, with different privileges granted with differing passnumbers.

Access matrix settings do not restrict functions like ⎕fnames and ⎕fnums. These functions never produce a FILE ACCESS ERROR because they do not work on a particular file. The ⎕fsize, ⎕fhist, or ⎕fstie functions have no limitations either; these are permitted if any other file operation is authorized.

Setting the Access Matrix

The ⎕fstac function sets the value of a file's access matrix. The syntax is:

```
    matrix ⎕fstac tieno
```

The left argument is an integer-valued, three-column matrix. A three-element integer vector is reshaped to become a one-row matrix. It replaces the previous value of the access matrix. Initially, when a file is created, its access matrix has no rows: its shape is 0 3. The file owner has complete access since the user number does not appear in Column 1 and no other user has any access. The following table shows the access code values.

Use the sum of all possible access codes to authorize all possible file operations. You can also use `¯1` to grant full access, as well as access to future file operations. A value of `9` indicates authorization to use the functions `⎕fread` and `⎕fappend` on the file. Any nonzero value permits use of `⎕fsize`, `⎕fhist`, and `⎕fstie`. **Note:** Not all combinations of values make sense.

Access Code Values

Code	Operation	Code	Operation	Code	Operation	Code	Operation
1	⎕fread	16	⎕freplace	256	Not used	4096	⎕frdac
2	⎕ftie	32	⎕fdrop	512	⎕frdci	8192	⎕fstac
4	⎕ferase	64	Not used	1024	⎕fresize	16384	⎕fdup
8	⎕fappend	128	⎕frename	2048	⎕fhold	32768	Not used
65536	Not used, represents ⎕fstatus on other sytems						

Reading the Access Matrix

`⎕frdac` function reads the access matrix into the active workspace. The syntax is:

result ← `⎕frdac` *tieno*

The result is the access matrix for the file tied to the number in the right argument. The second element in a row of the access matrix is the sum of the access code values for the particular file operations that are allowed for the user number in the first position of the row.

Changing the Access Matrix

In APL64, access to an APL component file is determined only when the file is tied. Consequently, changing the access matrix of an APL component file while it is tied has no immediate effect on access. The system uses the new value of the access matrix when another user tries to tie the file. Therefore, changing file access to prevent the file owner from erasing the file gives no protection until the owner unties or reties the file.

Under the rules of access control, you can be locked out of one of your own files. Since the ability to set the access matrix is one of the operations governed by the access matrix, you may be unable to correct the problem with `⎕fstac` alone. The `FILEHELPER` function in the `UTILITY` workspace allows you to access a file you own but cannot access.

Using Passnumbers

The third element in a row of the access matrix, the passnumber, is usually `0`. Nonzero passnumbers are used to exercise detailed control over file access. If the third element contains a passnumber other than `0`, the user whose number is on that row must provide a matching passnumber to perform file operations.

Omitting a passnumber from an argument is equivalent to providing an explicit passnumber of `0`. A mismatching passnumber causes a `FILE ACCESS ERROR`.

The passnumber used to tie a file remains associated with the file tie. After you tie a file with a passnumber, you must supply the same passnumber in all subsequent operations on the file. Using another passnumber produces a `FILE ACCESS ERROR`, even if some row of the access matrix happens to grant the appropriate permission with that passnumber. To use the permission granted by a different row of the access matrix, you must retie the file using that row's passnumber.

Passnumbers are intended for use within locked APL functions that are used in place of standard file functions. Suppose each component of the personnel file `2345 PERS` is a vector holding an employee's telephone extension and room number. The user whose number is `9876` is allowed to retrieve the telephone and room number, but not to change it. You choose a passnumber, `10349`, and define these locked functions:

```
    ∇PTIE
[1]   'C:\Production\PERS.SF' ⎕xfstie 25 10349
    ∇
    ∇R←PREAD N
[1]   R←2↑⎕fread 25,N,10349
    ∇
```

Next, you set the access matrix for `PERS.SF` to authorize read-only access by user `9876`. The row of the access matrix is:

```
9876 1 10349
```

Finally, you give the locked functions to user `9876`, but you do not reveal the passnumber. Now, user `9876` can tie and read from the file, but only by specifying the passnumber with each `⎕xfstie` or `⎕fread`. Since the user does not know the passnumber, access to the file `PERS.SF` is possible only through the functions `PTIE` and `PREAD`.

Using this technique, you can impose complex restrictions on file authorization; in fact, you can impose any sort of restriction you can state as an APL function. Since you can impose different passnumbers on different user numbers, you can provide multiple levels of access authorization to confidential data. The previous example showed access restricted to reading a file. Examples of other forms of control follow.

- Access only to even-numbered components.

  ```
  [1] ⎕ERROR (0≠2|N)/'EVEN NUMBERED CNS ONLY' ◇ R←⎕fread 25,N,32049
  ```
- Automatic logging of information requests.

  ```
  [1] A←N ⎕fappend 99 2888 ◇ R←⎕fread 25,N,32049 ◇ ⍝ LOG TO FILE 99
  ```

  Later use of `⎕fread` and `⎕frdci` on the file tied to `99` gives the value of *n*, the timestamp, and the requester's user number.
- Access only after validating data.

  ```
  [1] →(1≠⍴⍴V)/ERR ◇ →((20≠⍴V)∨0∨.>V)/ERR ◇ R←V ⎕fappend 45 14149
  ```
- Access only after verifying user identity by questions and answers (to protect a momentarily unattended computer from passersby).
- Access only to summary data (for instance, salary averages but no individual salaries).

Library Numbers

If you are maintaining legacy code, you may have to use library numbers. You associate a library number with a particular directory using the `⎕libd` command. If, for example, you created the files in program `SAMPLE1` in directory `C:\JOE\USR`, then the expressions

```
      ⎕libd '3 C:\JOE\USR'
      ⎕flib 3
```

produce

```
      3 PERSONS
      3 SALES
```

When you are using the traditional file functions that require a file id, you can use the library number and the file name in place of a full path name. Library number definitions do not affect extended ties. Note however, that strings consisting of a number followed by a name are valid filenames in some systems. They are accepted by the extended functions but may not have the effect expected. Consider the statements

```
'666 box' ⎕fcreate 9
'666 box' ⎕xfcreate 99
```

Both create component files. The first is in the directory defined for library `666` and has the name `box.sf`. The second is in the current directory and is seen by the Windows Explorer or File Manager as having the name `666 box`.

Colossal Component File System

Design Concepts

Basic Approach

This file system is similar to the long-established traditional component file system. Components are assigned in sequential order with each file append. Components can be dropped from the beginning or end of the file, but numbers at the beginning of the file are not reused. Information about each component is recorded in the component directory, which contains the component's location in the file and its length (among other things, including timestamp and user number). Currently, its component number is not explicitly written in the directory, but is implicit by virtue of the location in the directory.

When a component is replaced, the value is written to a new spot in the file. The space occupied by the component's previous data remains with the old data intact, but there is no pointer in the component directory to the location where it starts. Thus, this space becomes, at least temporarily, a "hole" in the file. The file system maintains a table of the locations and sizes of holes; the size of this table can be set as the second argument to `⎕cfcreate` or it defaults currently to `768`, which occupies three `4`K pages in memory.

When a component is either appended or replaced, it is located in the smallest hole into which it will fit; lacking any such hole, it is written at the end of the file. When the hole to be used is larger than the component, a remnant may be left, which remains in the hole table for future use. If the extra space is less than the size required to store a minimal component, including overhead information, the entire hole is allocated to that component and that scrap of space is unusable until the file is compacted.

If the hole table is full when a component is written to the end of the file, the newly created hole is written into the location of the smallest hole currently in the hole table, so that a minimum amount of space is rendered unusable (until the next time the file is compacted). If a hole is used completely, its entry in the hole table is set to `0 0`. This "bubble" is sorted out of the table the next time the file system looks for a usable hole.

File Sizes and Numbers of Components

The component directory has a theoretical maximum of `(2⋆32)-1` components, (something over `4` billion). The maximum file size is limited to a signed `64`-bit number. There is also a transaction counter; any operation that modifies a component file is considered a transaction, and each transaction is assigned a sequential transaction number in a `64`-bit counter. This facility provides a basis for historical tracing and better file recovery possibilities than in the traditional system. You could cause this counter to roll over by performing a million transactions per second for almost `600,000` years.

Programming Interface

In general, the new system functions you can use to manipulate this file system are parallel to the traditional and extended file system functions, but the overall system has been simplified. You use positive integer file tie numbers, and there is no interaction or checking for duplications with either the (old) traditional tie numbers or the extended file function tie numbers. The new functions will not work with traditional component files, and the existing functions will not work with these (Colossal) component files.

Corresponding Functions

You can create a file and tie an existing file exclusively or shared with `⎕cfcreate`, `⎕cftie`, and `⎕cfstie`.

You can read, append, replace, or drop components with ⎕cfread, ⎕cfappend, ⎕cfreplace, and ⎕cfdrop.

You can rename, erase, and untie files with ⎕cfrename, ⎕cferase, and ⎕cfuntie.

You can get information about a file or a component with ⎕cfsize, ⎕cfhist, and ⎕cfrdci.

You can get a list of tied files and file ties with ⎕cfnames, and ⎕cfnums.

You can lock a list of files with ⎕cfhold.

You can duplicate a file with ⎕cfdup which copies a file and allows a change in hole table size.

Functions with no Replacements

There are no equivalents in the new system for the functions ⎕fresize, ⎕fstac, or ⎕frdac. There is no provision for pass numbers or access limitation codes.

New Functions with no Correspondent in the Old System

You can see the version number of the running file system using ⎕cfversion.

The system function, ⎕cfinfo, provides information about the file. It takes the tie number as an argument and returns nine values: the file handle, the version number of the tied file, size of the current directory (total number of slots available for entries), number of directory entries occupied, number of slots in the hole table, number of active (available) holes currently in the hole table, the address of the first locking byte, the number of locking bytes available, and the active write synchronization value of the tied file.

The function ⎕cfsync requests the operating system to force pending data to the storage device, or sets the file synchronization value for the specified files. By default, file synchronization is done only when the file is untied. (However, this does not mean that all data are cached until then.) You can choose your times to invoke this function to optimize your performance and safety, or you can change the default to force synchronization on every write or never.

New Features with no Correspondent in the Old System

This file system implements a component zero, whose address is in the file header, but not the component directory. You can store any information in this component; it is intended for file identification. You write this special metadata component with '*data*' ⎕cfreplace tieno 0. You read it with ⎕cfread tieno 0. It does not affect the starting component number, and you cannot drop it. By default, it returns an empty character vector.

This file system has improved damaged-file recovery capability and data integrity checks, compared to the traditional component file system. These are accomplished by carrying additional information, called the prefix, with each component. The prefix comprises eight fields: a marker with the value 0xFFEEDD00; the length of the entire component, including the prefix and any trailing scrap space; the length of the component's data, which includes some identifying information, called the component header, that describes the data (and makes it an m-entry in our parlance); the component number; its timestamp; the transaction number (described above); and two 64-bit fields that contain a "message digest" of the data calculated with the widely published MD5 encoding algorithm.

Each m-entry (data and component header) is put through an MD5 hash calculation when the component is written. When a component is read, the digest is generated again and compared to the value stored in the prefix. A mismatch generates a File System Internal Consistency error. This should eliminate the already rare occurrences of APL crashes due to data corruption of components in a file.

The component length, timestamp, component number, and transaction number should be sufficient to rebuild a file if the component directory is damaged. You can use the transaction number to distinguish the current version of a replaced component from its historical predecessors.

Although the MD5 algorithm is in the public domain, the code for generating the message digest is copyrighted by RSA Data Security, Inc. and Gary McNickle. Specifically, the code is identified as the "RSA Data Security, Inc. MD5 Message-Digest Algorithm" and the C++ implementation of it is a derivative thereof. The APL64 implementation of it is likewise "derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm." License is granted by RSA to make and use derivative works provided that such works are identified as quoted in the previous sentence in all material mentioning or referencing this software or this function. RSA makes no express or implied warranty concerning the product.

As an adjunct to this algorithm, you can now use ⎕dr with a left argument of 'MD5' and a right argument of a simple array or scalar of any data type. This is further discussed in the description of ⎕dr in the System Function manual.