FilePlus
Named-Component File Extensions

Davin Church
Creative Software Design
July 30, 2023

Table of Contents

Introduction

This document describes a Dyalog namespace that contains a set of component file management routines written in APL by Davin Church of Creative Software Design for use in APL applications. Use of this code for any purpose is hereby granted, but credit (comments left inside the code, for instance) is requested and modification is not generally recommended (to allow for later updates). Dyalog APL v17.0 or later is required.

These functions implement a simple and high-performance method of using named components in Dyalog component files, allowing APL files to be used as a Key-Value Store (KVS). They are designed as extensions to the standard file system and may be used in conjunction with the native ⎕F... system functions. Both numbered and named components can coexist in the same file and operations may be performed interchangeably. Even if named components are not needed, these functions can still be used to manage dynamic component allocations by logically dropping components from the middle of a file.

Of course, most APL programmers have written similar tools over the years, sometimes several different sets of such tools. But many of them have often had significant restrictions, inflexibility, bugs, or efficiency concerns to deal with. This set of tools does not have any significant drawbacks of this sort and is well-suited for general use for all but the most extreme application needs. These functions are being distributed in the hopes that they may satisfy any and all future named-component file needs, without any need to "reinvent the wheel" when such facilities are desired.

The mere fact that many programmers have created such tools time after time for many different purposes points to their general usefulness. The reason they are not more widely used likely has a lot to do with the bother in producing a useful and efficient system while in the midst of writing an application. With tools like these readily available they often make application coding easier in several respects, even when it isn't otherwise important enough to build a custom system for those needs.

Overview

The following functions are available for use:

IO	Read, replace, or append a named (or numbered) component.
Dir	Perform named-component directory management, if needed.
Tie	Tie and untie files. (Only needed if special functionality is desired.)
Create	Create a new file. (Only needed if special functionality is desired.)
Sharing	Adjust global setting for automatic file sharing features, if needed.

The only required function for using named components is IO. It can accept physical component numbers as well as component names, and thus can replace ⎕FREAD, ⎕FREPLACE, and ⎕FAPPEND any time it may be desired.

Dir is commonly used as well for many applications. It can provide a list of all the named components in the file, determine if a named component exists, remove named components from the file, and perform other similar file management tasks. It can also be used to mark physical component numbers as reusable so that numbered components can be logically removed from the middle of the file. This feature is useful even when named components are not in use.

Tie is primarily a cover function for ⎕FSTIE, but provides other options such as detecting a currently-tied file and returning its tie number. It can also accept a file name without a full path and will refer to that name as being relative to the directory where the workspace is stored. Applications that may reside in various places may find this feature useful for portability reasons. If special features such as this are not needed then ⎕FSTIE may be used directly.

Create is a cover function for ⎕FCREATE, but it also allows "reserved" component numbers to be specified for this file. These components will never be used by IO and will remain permanently available for exclusive numbered use by the application. This feature may be used when upgrading an application that uses specific component numbers by adding named components to the file without disturbing the fixed-position components. If special features such as this are not needed then ⎕FCREATE can be used directly.

Sharing provides control over the automatic sharing features when single files are to be shared across processes or threads. ⎕FHOLD and :Hold are invoked automatically when the sharing control is non-zero. This is only needed if files are shared, and is typically only used once since the setting can be saved with the namespace. Note that automatic ⎕FHOLDs will conflict with any explicit ⎕FHOLDs in use at the same time and should not be used simultaneously with them. These ⎕FHOLDs and :Holds only persist for the duration of a single FilePlus operation and cannot protect a group of operations or a group of files. More complex ⎕FHOLDing and :Holding must be done explicitly.

IO and Dir can accept the usual file tie numbers for repeated use, but they can also accept a file name (suitable for use by Tie) for use on a file that may not already be tied. If it is not, the file will be tied, the operation performed, and the file untied. Of course, while this is convenient for single file operations, performance will suffer greatly if it is used frequently.

General Notes

Component names (see below for more details) can be almost any APL value except for empty arrays and simple scalars. Non-integer simple scalars are accepted but raveled before use. Component names that are simple character vectors are further processed for more convenient use but other names are used exactly as given. Therefore it is reasonable to use matrices, nested arrays, etc. of any reasonable size, shape, depth, or type as component names. (Floating point values are discouraged within a component name due to representational rounding difficulties.)

In addition, component names that are simple character vectors have been extended to allow the use of "array components". This means that a single name may be followed by a subscript notation (in traditional APL form) to create a component that is an item of a vector (or other array of any rank). These subscripts may be integers as usual for APL, but they need not begin at `⎕IO` nor do they need to be contiguous or consecutive. In addition, subscripts may themselves be names, and names and integers may be mixed within the same dimension.

Component arrays may be of any rank and any size and even extremely large component arrays are stored and accessed very efficiently with high-performance B-Tree data structures. In addition, an array and a non-array component can have the same base name, as can arrays of multiple ranks. In other words, components named `FOO`, `FOO[]`, `FOO[;]`, `FOO[;;]`, etc., can all exist in the same file at the same time.

This system also allows physical file component numbers to be released from use from the middle of the file. This can occur during internal operations or it can be performed explicitly by using the Drop Component operation (`'↓'`) of the `Dir` function (on either named components or physical component numbers). These released components are tracked and reused automatically by `IO`. A free component number may also be allocated in place of a new one when using `IO` to perform a logical file-append operation. This helps keeps data files from wasting too much space.

It is a design restriction is that these tools require the exclusive use of file component #1, so applications may not use that file component. All other components are dynamically assigned and `IO` will avoid all component numbers directly used by the application.

File passnumbers are not supported, but since these are seldom used this is not expected to be of any practical concern.

The application may `⎕FAPPEND` any components it wishes to the file and those will not be disturbed by these functions. (`IO` also provides an alternative to `⎕FAPPEND` and this feature can re-use any free-space components that might be available.) However, the application should not directly modify any other components that it has not explicitly appended or reserved or the file structure will be damaged. (An exception to that rule is that `Dir` may be used to determine the physical data component numbers in use by named components, which are then also available for direct use during their lifetime.) Otherwise, all `⎕F`... functions may be used on these files as desired.

Any file operation errors caused by incorrect application use (including referencing undefined component names or numbers) are reported in the usual APL way, with a ⎕SIGNAL that stops the code at the function call. When expected, such errors may be intercepted using :TRAP or ⎕TRAP or a D-fn error guard (::) and alternative action may be taken.

File safety is always of concern when using complex facilities such as this, but these functions have been written with that in mind and no data will be lost by an APL crash while they are executing. (A complete system failure will still need to be protected against by the appropriate Dyalog file properties and/or explicit disk flushing with ⎕FUNTIE 0.)

Shared Files

The FilePlus system is focused on high-performance file use in a single-threaded, single-user environment. If files need to be shared among multiple instances or multiple threads of an application, some additional work needs to be done to allow for appropriate sharing of files. This involves using ⎕FHOLD operations around <u>all</u> calls to IO and Dir when sharing between users or processes, or using :Hold operations around <u>all</u> calls to IO and Dir when sharing between threads of a single application. Of course, these additional requirements will use a little extra processing time and thus will reduce overall performance, but they are required to coordinate access to shared files in the same way as APL component files normally require.

These additional holds may be coded explicitly by the application if desired, or in simple cases they may be activated automatically by IO and Dir if the Sharing function is used to globally set the type of holding to be performed on shared FilePlus files. These automatic operations are recommended when simple sharing is necessary so that the application does not become burdened with excessive shared file management code. See the description of the Sharing function for more details.

Shared Resources

Using ⎕FHOLD or :Hold operations explicitly will allow multi-access file control to be coordinated among cooperating applications, but this does not extend the cooperative sharing beyond access to the referenced files. For a more general form of application sharing and exclusive access to resources, FilePlus provides an extra feature that corresponds roughly to a file-based "semaphore" facility. A Dir operation provides an atomic-level command that allows an application to claim ownership of an arbitrary resource in a controlled way. See the Dir function's Take Ownership (↑) operation for more details.

Components

Component Numbers

`IO` can also use conventional file component numbers and perform the same operations as `⎕FREAD` and `⎕FREPLACE` by using the same syntax as the native functions. Both `IO` and the system functions can be used with numbered components at the same time as `IO` is using component names for other components in the same file. Named and numbered components can coexist in the same file as long as the application does not use the native functions to modify components it has not created.

Component Names

Component names can be used for easy and efficient mnemonic access of file components. An application that uses component names is inherently very flexible, as new components can be added to the file without affecting any existing code that is already using that file. Component names also easily provide for widely heterogeneous data storage in a single file, whereas a component number-based file is more often used for only a single type of data. These functions provide for very flexible component naming in order to accommodate as many application designs as possible.

Arbitrary Values

A component name can be almost any non-empty APL data value. Simple scalars are reserved for special purposes, though. A scalar integer refers instead to an explicit, fixed component number, just like those used for the native `⎕F…` functions. In this way, `IO` may be used to directly replace `⎕FREAD` and `⎕FREPLACE` (and `⎕FAPPEND`, if `0` is used as a placeholder component number) throughout an application, if desired. A scalar character (or other non-integer scalar) value is allowed to be given as a component name, but this value is raveled to a one-item vector before it is used. This also simplifies use in code that may sometimes have a scalar value and sometimes have a one-item vector by treating them as the same name.

Other than that, any file-writable APL value may be used as a component name, including matrices, nested arrays, or other arbitrary structures, and those names may be of any reasonable size. Character data capitalization matters (exact matches only), but names may contain any Unicode characters. Additionally, special features are allowed in the typical case when simple character vectors are used as names (see below).

Component names are completely independent of the component contents, so the names need not be found anywhere in the component data, unless that is desired for application reasons.

Character (Text) Vectors

The most common form of component name is probably a character vector. As noted above, character scalars are raveled into a `1⍴` vector before processing. For convenience, leading and trailing spaces are then stripped from the name (and it may not then be empty). Spaces may still be included as an interior part of the name, which can make component names more readable, but be wary of consecutive spaces causing readability issues. In addition, any character vector ending in text enclosed in square brackets is treated as an array component (see below).

These special character-vector-only features are not expected to cause any undue hardship to most programmers and applications, and can be quite convenient in most situations.

If character-vector component names are formatted like subscripted array names, but are not intended as arrays, little harm will be done. At most the name will appear in the directory as an array name but I/O will be processed in exactly the same way as if it were not an array. If this is not acceptable behavior for some reason, it is also usually feasible to make slight changes to the component name so that it no longer follows the array naming conventions. For instance, extra open or close brackets could be included within the name, thus making it an illegal array name. Or the structure might be changed, perhaps enclosed or turned into a one-row vector, so that it no longer qualifies. Or perhaps an extra suffix could be added to the end of the name, such as a period or other symbol, so that it no longer ended with the closing bracket. Any of these methods would force the use of a non-array name if minor component name changes can be tolerated.

Array Components

As mentioned above, some component names may actually represent whole arrays of file components. Use of arrays simplifies application access to sets of related data items, just like arrays do in APL variables.

To make use of this feature, use a simple character vector as the array name and follow it by a subscript notation in square brackets ("`[]`") in the same way as APL traditionally subscripts variables. Between the brackets are one or more subscripts (see below for a more detailed discussion), each separated by semicolons ("`;`") to indicate a multi-dimensional array, in the same way that APL syntax is used to refer to array elements. Component names without such brackets are considered to be "scalar" components and are treated as any other component name rather than as arrays. Arrays may have any number of dimensions and may contain any number of elements and may be sparse (where not all consecutive elements or intersections are defined).

When specifying a single array element, every dimension of the array needs to contain a subscript value of some sort. This is the case with all named-component calls to `IO`. However, `Dir` can sometimes refer to entire arrays as a unit, such as when checking to see if a particular array name is defined in the file. In such cases the subscripts may be elided, although the brackets and any semicolons are still required in order to indicate the rank of the array name.

Even though `IO` supports access to arrays of components, it can still only refer to one element of the array at a time. If multiple elements need to be accessed then it must be accomplished with loops or Each ("¨") or some similar construct.

Subscripts

Numeric

Array subscripts may be numeric integers, just as in APL. However, they must be provided in character form as part of the character name (as text inside the brackets). Therefore, if any provided subscript is composed entirely of digits (up to 9 digits) then it is converted internally into a number and processed that way.

Using `Dir` to search for numeric subscripts can only be done by exact match rather than the sub-string searching that is used for non-numeric values, for instance. In addition, numeric subscripts are `⎕IO`-sensitive – they start with `0` or `1` as appropriate for the `⎕IO` in use at the time. Moreover, this value changes as `⎕IO` does, where `[5]` refers to the 5th element when `⎕IO=1` but the 6th element when `⎕IO=0`, and different subscripts may refer to the same file component in different environments, just like APL arrays.

Numeric subscripts are quite convenient for keeping lists of components that have no independent identity beyond their position in the list. Unlike APL, however, numeric subscripts do not have to be contiguous and their values may be quite large (`<1E9`).

Since there are few limitations on numeric subscripts, they can also be used to look up specific data values of many sorts. For instance, Gregorian dates (in the YYYYMMDD form) can be used as subscripts to directly look up the application data to which the date is attached. Many other kinds of meaningful integers might be directly useful to the application as well.

If the array subscripts are inherently numeric, but for some reason they need to be processed as text instead of numeric values, the values might be adjusted slightly to include some non-integer content. For instance, the number might be prefixed by an "X", or it might be suffixed by a period, or any other kind of adjustment might be made so that it's not composed entirely of (up to 9) digits.

Character

In addition to using traditional numeric subscripts, array components may also make use of names for subscripts. These names are just ordinary text of any reasonable length and character-content (except for a few special reserved symbols as noted below), specified within the usual subscript notation. Spaces and other symbols of any sort are allowed, although leading and trailing spaces are removed from each subscript for convenience. Any subscript text that does not qualify as a number (see above) is simply stored as character instead. This includes values that appear to be floating point numbers.

Mixed Numeric and Character

Not only is there a choice between numeric and character subscripts but they may be mixed in any combination desired. For instance, a 2-D matrix could use names for the first dimension and numbers for the second dimension. But any dimension can also be composed of both named and numeric subscripts simultaneously. Any combination within an array is possible and efficient.

Reserved Subscripts

As mentioned above, each array subscript value is required (it may not be left blank or empty), except in the cases of certain `Dir` functionality where it may be used to refer to entire arrays. Otherwise, almost any textual content may be used as the subscript, except as follows.

The characters "`[`", "`;`", and "`]`" may not be used as part of a subscript name because they change the meaning of the array syntax itself.

It is worth noting here that the subscript value of "`0`" is permitted, but treated differently in different environments. If `⎕IO=0` when it is used then it is a numeric subscript value (subject to `⎕IO` sensitivity). If `⎕IO=1` when it is used then it becomes a character subscript and is treated as an exact character value. Both may potentially end up in the array at the same time. This may be quite confusing in an application that uses different `⎕IO` values at different times.

Also, array subscript names of either "`⎕`" or "`,`" are currently reserved for anticipated future expansion, so you cannot use subscripts with the exact textual value of either "`⎕`" or "`,`" at the current time. Otherwise, subscript names that are longer but contain "`⎕`" or "`,`" can be used normally.

Likewise, any array subscript name that begins with the character "`⍙`" is also reserved for anticipated future expansion, so also avoid the use of "`⍙`" as a leading subscript character as well.

Usage Possibilities

Component Naming Suggestions

Adopting a naming convention for an application's file components can enhance code readability. Some possible uses and suggestions for names to use include:

- A persistent workspace variable that is preserved in a file between executions could use the same name in the file as the name of the variable in the workspace, thus providing a clearer logical connection.
- An important workspace value that is used only occasionally in different areas of an application is often stored in a global variable. To avoid using global values for this purpose and optionally to provide persistence (as above), the value could be stored in a file component using the same name as the global would be, and it could be kept neatly localized in the few places where it was needed.
- Named components could be used to store several separate items of general application configuration or control information (in "scalar" named components) at the same time as most of the file is used for a growing collection of application-data components (optionally in one or more large array components), without explicit partitioning of the file. Then if new items need to be added (which would otherwise be inserted between fixed positions causing renumbering) then that can be accomplished by just using a new representative name without any code changes such as fixed positions would require.
- A component name of "`⍺`" might make a good identifier for a general file-description component (especially if it's the first component written to a new file and so physically ends up in component #2).
- A naming convention similar to "`⍺xyz`" might be a good place to store any documentation or other descriptive information that a component named "`xyz`" might require.
- To store a function listing or a named variable in a code-repository style of file, a prefix or suffix character of "`∇`" or "`←`" might be an appropriate convention to use.
- Component names may be given implied structure, such as using `'MyClass.MyObject.MyProperty'` or `'dir/subdir/item'` or `'first, then, more, last'` or anything else that seems appropriate. Or they may given an explicit structure (e.g. `'Top' 'Group' 'Detail'`), since nested arrays can be used as component names. Note that these components are not stored in any special way internally – they're just single names just like any other component name. Use array components if a set of values collected together into a unit is needed.
- Use naming conventions with named array subscripts, too. For instance, use `Names[First;1]` and `Names[Last;1]` to store identifiable subsets of a list with mnemonic meaning. Named subscripts are also useful to reference identification information directly, of course, such as credit card numbers (e.g. 1234-5678-9012-3456) or real estate identifiers (e.g. SE-12-20-33-W1) or email addresses. A phone number might also be useful, especially as a secondary reference point.

- Named arrays with non-consecutive numeric subscripts may be convenient to use when the key to the element is inherently numeric (and `⎕IO` is consistent). For instance, dates of various formats (*YYYYMMDD*, *YYYYDDD*, or *JJJJJJJ*) might be useful ways to index data. Any sort of numeric data identifier (e.g. a person's government ID # or customer #, a product #, or an invoice #) also makes a good numeric index value.
- Any name (subscripted or not) can be mnemonically used for an exclusive claim on a shared resource, so that multiple applications can operate simultaneously without interfering with one another when accessing any particular application-defined resource.

Mixing Numeric and Named Components

Named components are excellent for storing identifiable items of data, but they don't do as well when there are long lists of data where meaningful names aren't available. Sometimes it is beneficial to store both named and numbered components together in the same file.

Example 1

For an example, let's assume that a number of clients have their transaction data stored in file arrays by their name, and each name contains a list of transactions for which numeric subscripts are used (sequential numbers or dates or transaction numbers, for instance). Each transaction then becomes a file component containing a variably-sized list of products in that transaction. But the information stored for each of these products is bulky and inconsistent so that each one needs to be stored in a separate file component. Now the problem becomes how to refer to these product components.

One reasonable solution is to put each product's information in a numbered component, without a name because any names here would just be artificially-created anyway. Instead, store all the product components and keep their physical component numbers in a vector in the transaction component. That way a client name array refers to multiple transactions, but each transaction's data refers to a list of physical component numbers directly.

Now the file contains a set of named arrays, each element of which eventually points to a set of unnamed components in the same file. Since the unnamed components can be managed as easily as the named components, this is now a hybrid file structure sharing the same disk space.

Example 2

Let's assume there is an existing application that already uses ordinary component numbers from 10 to 27, but extra room should be left for possible expansion. It is convenient to convert this to a FilePlus file so that named components can be added to the file structure for better application support. (Some of those numbered components may eventually be converted to use names as programming time permits.) Create a FilePlus file and specify components numbered 10-50 as reserved components and then use the file normally with both the old-fashioned component numbers and traditional system file functions and with the new named components accessed using `IO`.

Packaged Code or Data

In the early days of APL, when workspace sizes were very limited, programmers often kept large applications in "function files". These were most easily stored and accessed by referencing the file components by name, using the same name as the function it contained, with specialized code to manage the directory information explicitly. Of course this would obviously be easy to do with FilePlus, but while those original needs are long past variants of this idea can still be used for specific purposes.

For instance, functions might occasionally still need to be managed outside the workspace, often in "groups" or "packages". An array name could be used to indicate a package and its subscripts could be the names of the functions themselves.

Perhaps several alternate versions of a code group may be needed in different situations, such as when running on different operating systems (or other environmental condition). These alternate versions could be stored with names representing the appropriate operating system. The code needed for the correct operating system could be swapped into the workspace when the application is run. This might also be used at the namespace level to bring in alternate or additional namespaces as needed.

Or variables may need to be stored instead of functions. If several different application users are using the same system, their particular information may be stored in a named array on file and brought in for use, by user name, when the user "signs in" to the application.

Dynamically-updated variables could be tracked this way as well, such as monitoring stock market prices for thousands of stocks on a daily basis.

Documentation or other support information may need to be kept close to hand in named components.

The application may allow users to create and store their own related information, such as their photos on a subject and their comments on them. Named components would be a simple and natural way for the users to keep up with their data and update it as needed.

Sometimes programmers like to keep their toolsets in a common store and have their applications refresh themselves from the store each time they run, so they're always working with the most recent version of the code. Named -component files are a natural way to manage this need.

Whole workspace contents could be stored with named references, for transfer to another application such as those used to perform documentation, analysis, comparison, versioning, or other such needs. Two-dimensional arrays could also keep additional information alongside each function, such as ⎕AT timestamps or sequential version numbers.

Mini-Databases

A named-component file can operate similarly to a small database with many of the same indexing efficiencies, although accessing only one record at a time (which is reasonable for many small applications). Consider a FilePlus file that is to be used like a database table. The application data in this table could be stored as an array name (such as `Record[]` or `Data[]`) and each file component could contain a database record (in any form acceptable to APL and the application). The subscript used on the array name would be the equivalent of the table's primary (unique) key and could be used to identify a single data record in the table. By itself, this would be similar to the ancient ISAM indexing technology from mainframes, where any single record could be found directly by its key. If the data does not already have a field that would be natural to use as a unique identifier then a sequential integer may suffice. If using such a database file as if it were part of a relational database, then other tables could point to this one (a foreign key) by simply listing its primary key value within their data record.

However, a primary index on a database table is often insufficient to locate desired information. One or more secondary indexes may be needed to be able to find the appropriate records. To provide a secondary index simply create another array name (the index name) whose subscripts are the values to be indexed (names or dates or statuses, etc.) For example, `LastName[Smith]` could be an entry in the `LastName` index. When that component is read, it would contain a list of all the table's primary keys which contain data for people named "Smith". Those primary keys could then be used to read the correct data records directly from the data table.

For more complex queries multiple secondary indexes could be used at one time. For instance, looking up secondary indexes of `FirstName[Joe]` and `LastName[Smith]` would yield two primary-key vectors that could be combined together with Intersection ("`∩`"). The result would then be a list of primary keys of data associated with "Joe Smith". Any number of indexes may be combined in this way before ever reading the actual data from the file. A small routine could be written to syntactically simplify SQL-like processing of such combinations.

In addition, yet another named component (e.g. `Indexes`) could be present in the file that contains a list and description of secondary indexes that are defined for this table. This list could be accessed automatically by a generic subroutine whenever data records are written and it could automatically update all the table's indexes upon every write of a data record. This would keep all the indexes synchronized together much like a database system would do implicitly.

In a more unusual situation, a table may have a multi-part primary key but any individual portion of that key may need to be used independently. If the array were stored with multiple dimensions, one dimension per portion of the key, then `Dir` could be used to find full subscripts that match any combination of sub-keys and thus the actual records to which they refer. For instance, `Data[`*name*`;`*date*`;`*sequence*`]` might provide a structure for such an array.

Of course, this type of architecture would not provide all the facilities of an actual database system, but it's not an unreasonable way to keep track of simple data for a small, self-contained application that doesn't have a significant need for a separate database facility.

Alternative Names

Since component names and array component subscripts are case-sensitive, it may occasionally arise that the exact capitalization of a component or array subscript may not be easily available. For example, an array may be subscripted with the value "Davin" but it needs to be referenced as "DAVIN". If the capitalized "DAVIN" can't be used for the subscript in the first place, an extra reference can be accomplished in a roundabout way by creating another component array where the subscripts are stored in all capital letters and the contents of that array element tell how to capitalize the primary array subscript. Multiple variations on this theme are possible, including translation to multiple capitalizations at once or removal of spacing and punctuation for easier reference. Possibilities also exist for translation of non-array components, with some additional effort.

This mechanism can also be used in conjunction with the database-style storage mentioned above, to provide extra case-insensitive indexes or similar transformations.

Releasing (Dropping) Unnamed Components

Whether named components are needed in a particular file or not, the ability to make use of dynamic component numbers can be beneficial in itself. Many programmers during their careers have created methods to release unneeded component numbers from the middle of a file and reuse them again later when a new component is needed. FilePlus files can handle this directly without writing any extra management code.

Migrating Applications from Standard APL Files

An existing application using APL data files (with numbered components) may be incrementally converted into a FilePlus file reasonably easily. The first and most significant step is to move file component #1 elsewhere (using either a named or numbered component), because FilePlus files require that component as their base index component. Replace that component's data with an empty vector (`⍬` or `''`) and the next use of `IO` or `Dir` on that file will automatically begin using it.

All existing numbered components will be reserved for direct (numbered) use and so the application code need not have any changes there. Also, any new components that are `⎕FAPPEND`ed during the application are also automatically reserved for the application's direct use. The only restriction here is that the application should not assume that all appended components will be sequential (contiguous) because FilePlus may append some of its own components from time to time.

If desired for consistency in readability (and the ability to use any other FilePlus features), you may replace any or all calls to `⎕FREAD` and `⎕FREPLACE` with calls to `IO` using the same syntax. `IO` will add only insignificant overhead to the native calls when used this way. If desired, `⎕FAPPEND` may also be replaced with a call to `IO` by adding an additional "component number" parameter of `0` after the file tie number. For example, `X ⎕FAPPEND 1` could be replaced with `X IO 1 0` to achieve the same effect. (The returned component number is not shy in this case, but applications generally must capture the result anyway.) The only functional difference from `⎕FAPPEND` would be that an unused component from the middle of the file (if one is available) might be used instead of adding a new component at the end of the file, so the `IO`-appended number cannot be predicted. These changes may be applied instance by instance as desired without affecting the rest of the application.

After any application conversion steps are taken, new code may use either the traditional file functions to work with numbered components or `IO` can be used for access to either numbered or named components as desired. The application can be changed (one component number at a time) to use named instead of numbered components anywhere it would seem to be more convenient, while the application remains operational during the step-by-step upgrade process. The application might wish to use `Dir`'s "exists" (`∈`) functionality to determine if any particular data component has yet been changed from numbered to named usage, in order to support the dynamic conversion of data files.

Quick Reference

The FilePlus system provides five public functions for use, each of which has multiple syntax choices. When combined with the standard system file functions they provide all the tools needed for working with file components referenced by either names or numbers.

filetie ← *[reserved]* `Create` *filename*	⎕`FCREATE` a FilePlus file.
[result] ← *function* `Dir` *file [component]*	Perform file directory management.
data ← `IO` *file component*	Read a file component.
data `IO` *file component*	Add or replace a file component.
number ← *data* `IO` *file* `0`	Append a file component by number.
filetie ← *[options]* `Tie` *filename*	⎕`FSTIE` a file.
`Tie` *filetie*	⎕`FUNTIE` a file.
oldholdtype ← `Sharing ⍬`	Return current file-sharing method.
oldholdtype ← `Sharing` *holdtype*	Change current file-sharing method.

Major Parameters

file	May be either a file tie number or a full or relative file name.
component	May be a physical component number or a component name. Component names may be almost any APL value, as described in detail above, but are often just character vectors (with or without subscript notations).
function	A mnemonic character scalar or vector that specifies the directory operation to perform, selected from the following list: `?` Return a list of all existing component names. `∈` Determine whether the component (or array) exists. `⍳` Return the physical file component number for the component name. `[?]` Return a list of all defined subscripts for a given array. `[⍳]` Return a list of all physical component numbers for the named array. `[>]` Return the next array subscript immediately following the one given. `[<]` Return the previous array subscript immediately preceding the one given. `[+]` Return a list of defined subscripts beginning with the prefix given. `↑` Take control of a named component (claim semaphore). `↓` Drop a named or numbered component and free the component it used. `≠` Return a list of all user-reserved physical component numbers. `⍬` Return a list of all free-space physical component numbers. `⍰` Validate the file's directory structure and return a list of the application-created physical component numbers.

Syntax and Descriptions

Create

Create a new FilePlus file:

filetie ← *[reserved]* `Create` *filename*

`⎕FCREATE` a new file, initialize it as a FilePlus file, and return the file's tie number. Unlike `Tie`, the file is left exclusively tied.

The filename argument may omit a full directory path (no root directory specified) to indicate that the file is to be created relative to the current workspace's (`⎕WSID`) directory.

If `⎕FCREATE` is used instead of `Create`, the internal FilePlus directory will be automatically initialized upon its first use by `IO` or `Dir`. So that this automatic initialization can be performed, leave the file empty before this first use, or append file component #1 as a `''`. File component #1 is always required for internal use by the FilePlus system.

Optional use

Use of this function is not required unless an optional list of pre-reserved component numbers is specified during creation or a relative-path file name is being supplied.

Examples

```
tn←Create 'C:\App\Data.dpf'
tn←Create 'Data.dpf'
tn←2 3 4 5 Create 'C:\App\Data.dpf'
```

Dir

Perform various FilePlus directory-management functions:

[result] ← *function* `Dir` *file [component]*

The *file* parameter may be a `Tie`/⎕`FSTIE` file number or a file name suitable for passing to the `Tie` function. Use of a file name (that is not already tied) will tie and untie that file with each use. Use of pre-tied numbers is most efficient for multiple calls. See the `Tie` function for more details on using a *file* name.

Functions

Functions are specified as character codes as the left argument of `Dir`. Choices are as follows:

`?` — List Components

Return a list of all component names in the file. Array names in the list are suffixed with empty subscript notations (they include the brackets and semicolons but no subscripts) in order to indicate the rank of the array. Names are returned in sorted (⍋) order.

The *component* name in the right argument is optional, but may be included to specify a partial component name on which to search. If present, only component names containing (⍷) that partial name are returned in the result.

`∈` — Does Component Exist

Return a Boolean indicating whether or not that component is defined in the file. A *component* name is required in the right argument.

If the argument specifies an array name, it must include the brackets and semicolon to indicate the rank of the array. It may optionally contain a specific subscript of that array. If a subscript is specified then the existence of that element is checked. If the subscript is empty then the whole array (of the indicated rank) is checked to see if the array exists at all. Partial subscripts are not accepted as input for this function.

`⍳` — Physical Component Number

Return the physical component number in use for the *component* name specified in the right argument. A full component name must be specified, although it may be a specific element of an array. Array names without subscripts are not accepted.

This is useful when the application needs direct access to the physical file component, such as when ⎕`FRDCI` needs to obtain component information.

[?] — List Array Subscripts

This is only used on a component array. It returns a list of all the defined subscripts on that array in sorted (⍋) order. The *component* name in the right argument must be an array name and it must include brackets and semicolons to indicate the desired rank of the array.

The array *component* name may optionally include partial subscript items within the brackets. If it does so, then only subscripts that contain (∈) that text within the subscript (each dimension must match separately) will be included in the result. (Numeric subscripts require an exact match.) For example, requesting `FOO[7;]` will return all the defined subscripts on row `7` of that component matrix.

The shape of the result changes depending upon the rank of the requested array. If the array is a vector, then the result is a vector of single subscripts (numbers and/or nested character vectors). If the array has multiple dimensions then the result is an *n*-column matrix for an *n*-dimensional array.

[⍳] — Physical Array Component Numbers

This is only used on a component array. It returns a list of all the physical component numbers used by all the elements of an array, in the same order as the "`[?]`" function would return them when given the same *component* name argument. The *component* name in the right argument must be an array name and it must include brackets and semicolons to indicate the desired rank of the array.

This is useful when the application needs direct access to the physical file component, such as when `⎕FRDCI` needs to obtain component information.

The array *component* name may optionally include partial subscript items within the brackets. If it does so, then only subscripts that contain (∈) that text within the subscript (each dimension must match separately) will be included in the result. (Numeric subscripts require an exact match.) For example, requesting `FOO[7;]` will return a vector of all the component numbers used by the defined elements on row `7` of that component matrix.

[>] — Next Array Subscript

This is only used on a component array. It returns the succeeding defined subscript, in sorted (⍋) order, in use by that array. The *component* name in the right argument must be an array name and it must at least include brackets and semicolons to indicate the desired rank of the array.

If a specific subscript is named in *component*, whether it exists or not, then the succeeding subscript in order is returned as the result. If the entire subscript is left empty (e.g. `[;]`) then the first ordered subscript is returned. Partial subscripts are not accepted as input for this function.

The shape of the result changes depending upon the rank of the requested array. If the array is a vector, then the result is simple (a number or a single character vector). If the array has multiple dimensions then the result is a vector with one item per dimension. If there is no succeeding subscript, the returned result is ⍬.

⎕< ⎕ — Previous Array Subscript

This is only used on a component array. It returns the preceding defined subscript, in sorted (⍋) order, in use by that array. The *component* name in the right argument must be an array name and it must at least include brackets and semicolons to indicate the desired rank of the array.

If a specific subscript is named in *component*, whether it exists or not, then the preceding subscript in order is returned as the result. If the entire subscript is left empty (e.g. `[;]`) then the last ordered subscript is returned. Partial subscripts are not accepted as input for this function.

The shape of the result changes depending upon the rank of the requested array. If the array is a vector, then the result is simple (a number or a single character vector). If the array has multiple dimensions then the result is a vector with one item per dimension. If there is no preceding subscript, the returned result is ⍬.

⎕+ ⎕ — List Array Subscripts With Prefix

This is only used on a component array. It returns a list of all the defined subscripts on that array in sorted (⍋) order. The *component* name in the right argument must be an array name and it must include brackets and semicolons to indicate the desired rank of the array.

The array *component* name usually includes partial subscript items within the brackets. If it does so, then only subscripts that begin with (⊃⍷) that text within the subscript (each dimension must match separately) will be included in the result. (Numeric subscripts require an exact match.) For example, requesting `FOO[My]` will return all the defined subscripts beginning with the text "My" in that component vector, such as "MyName", "MyAddress", and "MyPhone".

The shape of the result changes depending upon the rank of the requested array. If the array is a vector, then the result is a vector of single subscripts (numbers and/or nested character vectors). If the array has multiple dimensions then the result is an *n*-column matrix for an *n*-dimensional array.

↑ — Take Control of Component (Claim Semaphore)

This *function* is only used in shared-application environments and otherwise serves no practical purpose. Its goal is to allow an application to claim exclusive control of a given component name, which nominally represents some sort of application resource. It can be considered a general form of a file-based semaphore or Mutex facility. While ⎕FHOLD can request exclusive access to a whole file, this logical semaphore operation can be used to claim access to any arbitrary resource (as defined by the application). However, this *function* is only operationally valid if the shared file is being held (with ⎕FHOLD or `:Hold`, as appropriate) while control is being claimed, like any other *function*. It does not replace any need for such file-level holds when using `IO` or `Dir`.

Any component name, subscripted or not, may be used as an application's mnemonic resource semaphore. When a component with that name exists, the semaphore is considered to be in use. When that component name does not exist, the semaphore is available for use. Only applications that obey these semaphore rules are cooperating in access to the resource.

To attempt exclusive access to the semaphore, issue `Dir`'s `'↑'` (Take Control) function with the name of the component (the semaphore name). If a component with that name does not exist (the semaphore is available), then it will be instantly (and atomically) defined and a `1` will be returned as a result from `Dir` to indicate that the semaphore has been acquired. If the component already exists, then nothing will be changed and a `0` will be returned as the result to indicate that the semaphore is already in use and is not available. This is functionally similar to a Test-and-Set instruction in computer hardware architecture.

An application with exclusive semaphore access should release the semaphore as soon as it is no longer needed. To release control of the semaphore component, issue a Dir `'↓'` (Drop Component) function (see below) to delete the component from the file. The application is responsible for managing any "lost" semaphores (due to a crashed application, for instance) in whatever way it desires (such as timeouts, resource in use confirmation, or content invalidation).

While the application has exclusive access to the semaphore component it may store any data desired in that named component and use it in any desired way. The only restriction is that the component may not be deleted, as that signals the release of the semaphore. Many semaphores may be acquired and managed simultaneously by an application.

↓ — Drop Component

Delete the named or numbered component from the file and mark it as available for re-use by `IO`. This includes the function of releasing a semaphore component that was claimed by the Take Control operation, above.

If a physical component number is provided, it must be one that the application has previously created with `IO...0` or `⎕FAPPEND`. That component is released from use and is now explicitly available for future allocation by either named or unnamed components. This feature may be used for dynamic component management even if named components are not in use. This functionality may require additional validation time if the file also contains large named-component-arrays.

If a component name is provided, it should be a full component name. If it names an array then it must at least include brackets and semicolons to indicate the rank of the array. If it further includes an exact subscript naming an element of that array, then only that one element is removed. If the subscript notation is empty (e.g. `FOO[;]`), then the **entire** array (of that rank) is deleted as a single unit (the array name and all its elements; caution is recommended). Partial subscripts are not accepted as input for this function.

Note: If the final remaining element of a named array is deleted then the array name itself is also removed from the file.

≠ — List Reserved Components

Return a list of all the physical component numbers that were reserved by the application during file creation (with `Create`). No *component* argument is permitted with this function.

⌀ — List Free Components

Return a list of all the physical component numbers that have been marked as deleted and reusable.

Deleted components are simply abandoned in place (for performance reasons) so they will continue to contain arbitrary data until reused. If many deleted components remain unused then they may take up a noticeable amount of file space. On the rare occasion that excessive wasted space should be removed, this list provides a map of unused components that may be specifically emptied of data (with `IO` or `⎕FREPLACE`) prior to a file data compression with `⎕FRESIZE`. Use caution that only these component numbers are rewritten. No *component* argument is permitted with this function.

⍰ — Validate Structure

This function performs a detailed analysis of the file's internal named-component structure to make sure that no logical errors or idiosyncrasies are detected anywhere. No *component* argument is permitted with this function.

In addition, it returns as its result a list of all component numbers that have been explicitly created by the application without names (using `IO...0` or `⎕FAPPEND`) and are thus not included in the named directory trees.

An application abort during a FilePlus operation will not damage the file structure, but it may leave unused components in the file if an operation was not completed. The result of this validation can detect and report such components as explicitly unnamed component numbers. If these can be distinguished from true application-created components, then `Dir` may be used to discard (Drop) them so they will be reclaimed for normal use.

This validation step does not perform a system `⎕FCHK` operation. If damage is suspected at the Dyalog file system level, such as after a full-system crash during operation, then `⎕FCHK` should be used before tying the file.

Examples

List of component names ***containing*** the string “name”:

```
'?' Dir 'data.dpf' 'name'
```

Does a component exist with the name “First name”?

```
'∊' Dir 123 'First name'
```

Does a component exist with the subscripted name “Name[First]”?

```
'∊' Dir 123 'Name[First]'
```

List all subscripts in component vector “Names[]”:

```
'[?]' Dir 123 'Names[]'
```

What is the physical file location of the component named “First name”?

```
'⍳' Dir 123 'First name'
```

Claim exclusive control of the “mine” application resource:

```
'↑' Dir 123 'mine'
```

Remove component “First name” from the file:

```
'↓' Dir 123 'First name'
```

Remove subscript “First” from component vector “Name”:

```
'↓' Dir 123 'Name[First]'
```

What is the next subscript (in ⍋ order) in the vector “dictionary” after the one named “water”:

```
'[>]' Dir 123 'dictionary[water]'
```

What are all the subscripts in the vector “dictionary” which begin with “wa”:

```
'[+]' Dir 123 'dictionary[wa]'
```

Return the list of user-reserved component numbers (defined during `Create`):

```
'≠' Dir 123
```

Return the list of all internally-unused component numbers (rarely useful):

```
'⍬' Dir 123
```

Validate file consistency and return numbered-only (e.g. `⎕FAPPEND`ed) components:

```
'⍰' Dir 123
```

IO

Read a named (or numbered) file component (that must exist):

data ← IO *file component*

Write (create or replace) a named (or numbered) file component:

data IO *file component*

Write an unnamed file component and return its physical component number (i.e. ⎕FAPPEND):

number ← *data* IO *file* 0

The IO function reads and writes named or numbered file components.

The *file* argument may be a tied file number or a file name suitable for passing to the Tie function. Use of a file name (that is not already tied) will tie and untie that file with each use. Use of pre-tied numbers is most efficient for multiple calls. See the Tie function for more details on using a *file* name.

The *component* argument is a component name or a physical component number; see the full description under "Components" on page 7.

- Reading a non-existent component results in an APL error.
- Writing to a new component name adds the data to the file.
- Writing to an existing component name replaces the existing component.
- Writing directly to a non-existing component number succeeds by extending the file as necessary.
- Writing a new unnamed component (with a 0) does not use a name and returns the physical component number assigned. This is identical to ⎕FAPPEND except that it may re-use an available (free) component number.

Since IO can be efficiently used with physical component numbers, using it in place of ⎕FREAD, ⎕FREPLACE and ⎕FAPPEND throughout an application provides additional consistency and readability to application code. Also, IO can be used to write to an explicitly- or implicitly-reserved component number for the first time (before the component exists in the file), which cannot be done with ⎕FREPLACE.

Writing an unnamed component (using either IO...0 or ⎕FAPPEND) implicitly reserves that component for use by the application as a numbered component (optionally via ⎕FREAD or ⎕FREPLACE) and it is thereafter avoided by these functions. Using IO...0 logically does the same thing as ⎕FAPPEND, except that it may instead assign an existing component number that is no longer in use if one is available. Be sure to have the application keep track of any returned component number because there's no reliable way to identify them later (just as native ⎕FAPPEND has always required).

An unnamed component that is no longer needed, even one in the middle of the file, can be released back to the "free component pool" (i.e. made available for re-use) by using `Dir`'s Drop Component ("↓") function on that component number. When combined with `IO`'s Append Component feature, this provides an easy way to manage dynamic component allocations without using named components at all.

Examples

Writing and reading with relative file name:

```
'Davin' IO 'Data.dpf' 'First name'
fname←IO 'Data.dpf' 'First name'
```

Writing with file tie number:

```
'Church' IO tn 'Last name'
```

Reading with absolute Windows file name:

```
lname←IO 'C:\App\Data.dpf' 'Last name'
```

Appending an unnamed component:

```
newcmp←directdata IO tn 0
```

Writing a component with a nested name value:

```
xyzzy IO 'Data.dpf' ('weird' (2 3⍴⍳6) ('cmp' 'name'))
```

Subscripted names are efficient and don't have to be contiguous:

```
'Davin' IO 123 'First name[1]'
'Church' IO 123 'Last name[777]'
```

A numbered matrix of people's names:

```
'Davin' IO 123 'Names[1;1]'
'Church' IO 123 'Names[1;2]'
```

A named vector of a person's names:

```
'Davin' IO 123 'Name[First]'
'Church' IO 123 'Name[Last]'
```

Mixing named and numbered subscripts in separate dimensions:

```
'Davin' IO 123 'Names[1;First]'
'Church' IO 123 'Names[1;Last]'
```

Sharing

To change the current global automatic file sharing setting:

oldholdtype ← `Sharing` *holdtype*

To query the current global automatic file sharing setting:

oldholdtype ← `Sharing 0`

FilePlus seems likely to be needed more often for small single-user applications so file sharing is not a primary optimization criterion. After all, a significantly multi-user application is more likely to be using a commercial database of some sort, which is probably better in those circumstances for many reasons. However, FilePlus is capable of handling component file sharing in a reasonable fashion when that is needed. It is important to note that FilePlus always refers to its complex internal control information any time `IO` or `Dir` is used, and this may not be changing during its execution. Therefore, all named-component calls (even read-only calls) to these functions should be surrounded by file-holding mechanisms in a shared-file environment.

A global shared-file setting may be specified by providing a new *holdtype* value from `0` to `3`. The previous setting value is returned as a shy result.

The setting values to be used are:

`0`	Files will not be shared and therefore no automatic holding is done (the default).
`1`	A `⎕FHOLD` will be performed around each call to `IO` and `Dir`.
`2`	A `:Hold` will be performed around each call to `IO` and `Dir`.
`3`	Both a `⎕FHOLD` and a `:Hold` will be performed.

This setting is stored in an internal namespace-global variable and thus will be saved with the namespace. Therefore, such a saved workspace does not need to reactivate the sharing setting dynamically at run time, although it does no harm to do so.

Holding a file during access is necessary when sharing a FilePlus file across multiple users, processes, or threads. When only one process/thread is accessing a file at a given time, then no holding is necessary. The default value of `0` signifies this situation and allows FilePlus operations to proceed at their maximum speed without concern about interference from interacting processes.

A value of `1` (or `3`) indicates that the files will be shared across different processes (including multiple users or copies of the application) and a standard file system `⎕FHOLD` will be used for access coordination by `IO` and `Dir`. This can slow down I/O processing due to the need for inter-system locking, but it's necessary when the files are being shared. (However, `⎕FHOLD` uses no time for files that are exclusively tied and it won't be used for direct raw-component-number I/O which is an atomic operation.) Automatic use of `⎕FHOLD` will also conflict with any explicit use of `⎕FHOLD` that persists across any call to `IO` and `Dir`, so combine use of this value and native `⎕FHOLD` calls with great caution. This use will also only apply to a single file at a time, so multiple files cannot be coordinated automatically. For any other, more complex needs, do not use the automatic feature and perform all `⎕FHOLD`s explicitly. `Dir`'s `'⍭'` (Take Control) function, when used while the file is held, may also be useful to assist with larger-scale shared-resource synchronization.

A value of `2` (or `3`) indicates that the files will be shared across different threads (those created by the "`&`" operator) that are executing within a single process. `IO` and `Dir` will temporarily use hold tokens partially named after the current file tie number and thus are unlikely to conflict with any use of `:Hold` elsewhere in the application. This mechanism operates much faster than `⎕FHOLD` and shouldn't degrade performance noticeably. However, since the tokens use a file tie number (partially for performance reasons) this does not account for different threads tying and untying the same files with different numbers (which is dangerous in any case). If the threads do not share consistently common tie numbers, do not use this mechanism and perform explicit `:Hold` operations instead.

These different types of holding are necessary in these specific circumstances and may not be used interchangeably. Using the wrong type of sharing hold will be ineffective and file damage will then be likely.

If automatic shared-file holding is not appropriate due to technical difficulties then this setting should be left at `0` and all synchronization must then be managed explicitly by the application.

Tie

Tie a standard or FilePlus file:

filetie ← *[tienumber [exclusive [writable]]]* `Tie` *filename*

Untie a standard or FilePlus file:

`Tie` *filetie*

`⎕FSTIE` or `⎕FUNTIE` an APL component file. If a file name is provided this will tie the file to the next available file tie number. If one or more numeric file tie numbers are provided, this will untie them.

Tying a file

When providing a file name, it will be share-tied to the next available tie number, which is then returned as the result.

The filename argument may omit a full directory path (no root directory specified) to indicate that the file to be tied is relative to the current workspace's (`⎕WSID`) directory.

If the named file is already tied then the existing tie number will be returned (rather than creating a new one), after turning it into a negative value. To use this negative number with the native `⎕F…` functions, first negate it or take its absolute value. The fact that it's negative can later be used to determine if the file was previously tied so that utility-style code (including `IO` and `Dir`) may avoid untying it when it finishes.

Advanced tying

If an explicit tie number is needed, or the file should be exclusively tied, or confirmation is needed that the file is not in a read-only state, an optional left argument of one to three positional values may be supplied, as follows:

[1] Normally, the file is tied to the next available number (which is returned). But if this parameter is provided with a valid file tie number and the file is not already tied, an attempt will be made to tie it explicitly to the given *tienumber*. [0=default]
[2] If true (`1`), indicates that the file should be exclusive-tied instead of share-tied (`0`, the default). If the file is already tied, no changes will be made.
[3] If true (`1`), indicates that read/write access to the file is mandatory. An error will be signaled if the file is read-only on disk. If false (`0`, the default), no checking is done.

Optional use

This function is only needed for tying if:

- A relative file name is being supplied
- Detection of an already-tied file is desired
- Detection of a read-write file is needed

…otherwise a simple `⎕FSTIE` (or `⎕FTIE`) is sufficient.

However, since the above features do not imply the use of `IO`, `Tie` may be used on standard component files without any use of named components. This allows the above features (relative naming, etc.) to be used on otherwise normal files.

Untying a file

To untie a file, simply provide one or more file tie numbers in the argument. These are simply passed to `⎕FUNTIE` after quietly ignoring any illegal or not-tied numbers.

Examples

```
tn ← Tie 'C:\APP\Data.dpf'
tn ← Tie 'Data.dpf'
tn ← ⍕17 1 0 Tie 'C:\APP\Data.dpf'
Tie tn
```

Internal Settings

FilePlus uses an internal (namespace-global) variable to adjust its operation, but this is not usually visible to the developer nor is it expected to ever be changed. But it is mentioned here in case special circumstances call for its use. If a change is needed, it is recommended that it be changed once during development and the new value saved with the workspace.

Note that if FilePlus is attached to your application as a direct namespace (e.g. #.FilePlus), then this setting should be referred to within that context (e.g. #.FilePlus.TreeSize). However, if FilePlus is attached indirectly using the Tatin package manager (although still referred to as #.FilePlus) then the context is somewhat different. Tatin uses a sub-level of a redirected namespace to provide limited access to the public functions and thus referring to internal settings must account for that extra nesting (e.g. #.FilePlus.##.TreeSize). The extra work required to access such settings is intentional in order to dissuade programmers from making changes to it without good reason and possibly causing considerable performance degradation by accident.

Note also that a few user-defined operators are also defined in the native FilePlus namespace, but those are for internal use only and are not available to the application programmer. Only the five public functions described (and the namespace variable below, if necessary) should be accessed by the application.

TreeSize

The entire FilePlus internal data structure is based on a high-efficiency data architecture known as a B-Tree and the primary control parameter for a B-Tree is the size of each node. The selection of node size depends on a performance tradeoff between the depth (the number of layers) of the tree, which affects the total number of index components in the file and the total number of index reads needed to locate data, and the size of each index component, which affects the time it takes to read and write each of those components (transferring the data to and from the disk).

A good compromise seems to be a *maximum* of about 100 reasonably-sized keys in each tree node, so that is the default value for `TreeSize`. However, using a large number of very large component names or needing to deal with an external constraint such as minimizing the number and access rate of index components may require a different node size for a better balance in your application. To this end, the `TreeSize` namespace-global variable may be changed to reflect those needs. For best results, this value should at least remain constant (for any given file, if not for the entire application), but if it is changed then the file's B-Tree structures will adapt to the new node size gradually during use.