Prospero Library Manual
Version 5

Draft of 30 March 1994
Document Revision No. 0.5.3

B. Clifford Neuman   Steven Seger Augart

Information Sciences Institute
University of Southern California

A digital copy of the latest revision of this document may be obtained through Prospero as /papers/subjects/operating-systems/prospero/doc/library.PS.Z, in the #/INET/EDU/ISI/swa virtual system, or through Anonymous FTP from PROSPERO.ISI.EDU as /pub/prospero/doc/prospero-library.PS.Z
This work was supported in part by the National Science Foundation (Grant No. CCR-8619663), the Washington Technology Center, Digital Equipment Corporation, and the Defense Advance Research Projects Agency under NASA Cooperative Agreement NCC-2-539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any of the funding agencies. The authors may be reached at USC/ISI, 4676 Admiralty Way, Marina del Rey, California 90292-6695, USA. Telephone +1 (310) 822-1511, email info-prospero@isi.edu.

Contents

1 WARNINGS

1. This manual is preliminary. It does not fully describe all of the Prospero library calls.
2. This manual reflects Prospero releases Alpha.5.2 and later. You can use it with Prospero releases between Alpha.5.0 and Beta.5.1; changes are documented under the appropriate functions.

2 Introduction

This manual describes the entry points to the Prospero library.

3 PFS Library

3.1 Introduction

The PFS library includes procedures for allocating and freeing Prospero data structures, resolving names using Prospero, reading directories, retrieving attributes, adding and deleting links, and creating directories. Only those procedures generally used by application programmers are described here. The remaining routines are called internally.

These functions are prototyped in the include file `pfs.h`, found in the `include` directory in the Prospero source tree. The data structures manipulated by these functions and the definitions of the flag options are also defined and documented in that file. When Prospero is installed normally, the library (ready to be linked with) can be found in `lib/pfs/libpfs.a` in the Prospero source hierarchy.

Since this manual is still not complete, a programmer will often find it helpful to look at the source code files for the functions discussed here. Most of them are preceded with large comments discussing their behavior in more detail than is gone into here. They can all be found in the `lib/pfs` directory in the Prospero source tree.

Examples of using these functions can be found in the `user` directory of the Prospero source tree.

Programmers looking for examples of listing a directory and retrieving attributes should look at `vls.c`. Examples of setting attributes are in `set_atr.c`. An example of the `pget_am` interface to retrieving files is in `vget.c` (after you see it, we expect you'll appreciate the simplicity of **pfs_open()**. An example of `add_vlink()` is in `vln.c`.

3.2 Multi-threading

Work to make the Prospero library safe in a multi-threaded environment, such as pthreads, is in progress. Since one of the Prospero developers shot off his mouth and promised to get a multi-threaded Prospero server to a certain loyal Prospero customer well before November 15th, you can expect this item soon.

3.3 Error Reporting

Most functions in this library return a numeric error code (defined in `perrno.h`). The functions in this library which return pointers to structures will return a null pointer in case of an error. They will indicate which error occurred by setting the global variable `perrno` (defined in the include file `perrno.h`) to one of the constants defined in that file. Note: Functions that return an explicit numeric error code are not guaranteed to set `perrno`.

Most functions that set perrno or return an error code will also set an explanatory message in the global variable `p_err_string` (also defined in that file). Even if they have no additional information, they are supposed to set `p_err_string` to the empty string so that the user isn't misled by an old error message.

The Prospero library guarantees that `p_err_string` will always be a pointer to a string of some sort, frequently empty. You don't have to worry about it being a NULL pointer. `p_err_string` and `p_warn_string` may change in value during the program's execution. If you are operating in multi-threaded mode, their values will be different in different threads.

The error indicators in `perrno` and `p_err_string` are persistent across library calls. To clear them before making a library call, use the function **p_clear_errors()**:

```
extern void p_clear_errors(void);
```

3.3.1 WARNING

All the functions in this library are supposed to obey the above convention about `p_err_string`. We have not gone over all of them to make certain that the convention is obeyed in every case, due to the press of other work.

3.4 Argument modification

A function that accepts a *VLINK* may modify that link if the object it points to has been forwarded. If it does so, it will modify the link to refer to the new location of the object.

Not all calls to the Prospero library currently do this. When we take the time to revise the library, we will attempt to make this behavior consistent across all functions in the library that accept *VLINK*s and send Prospero protocol messages that inform them the link's target has been forwarded.

3.5 Initialization

Every Prospero client **must** call the function **p_initialize()** before making any calls to Prospero library functions. Your program may well crash weirdly if you don't.

```
extern void p_initialize(char *software_id, int flags,
                         struct p_initialize_st * arg_st);
```

If you're writing a client, send email to `pfs-administrator@isi.edu` and request a software ID string. If you're just experimenting with the libraries, you can pass a NULL pointer or empty string as the software ID. At the moment, no flag values are defined, and no members of the structure for additional arguments are defined, so the most common calling sequence is:

```
p_initialize((char *) NULL, 0, (struct p_initialize_st *) NULL);
```

3.6 Data Structures

Most of the data structures in the PFS library are allocated and freed with special allocation functions (*VLINK*s are allocated with **vlalloc**, etc. The allocation functions also initialize the members of the structure to common values and the freeing functions de-allocate allocated memory referred to by the members of the structure.

Do not use the C library function **free** to free memory allocated by one of the special Prospero allocating functions, and do not use one of the special freeing functions to free memory allocated by **malloc**.

The *VDIR* structure does not have special allocation and freeing structures. This is because in the current uses of the Prospero library, one generally does not make linked lists of *VDIR* structures; instead, one allocates them on the stack with code that looks like this:

```
VDIR_ST dir_st;
VDIR dir = &dir_st;
```

and then initializes them with:

```
vdir_init(dir);
```

and frees the allocated memory referred to by the members with:

```
vdir_freelinks(dir);
```

3.6.1 Linked Lists

Prospero uses linked lists of items as a basic data structure. In all cases, these are doubly linked lists. As usual, the empty list is represented by a list whose head is the NULL pointer. When a doubly linked has one or more items in it, then the `previous` field of the first item in the list points at the last item, and the `next` field on the last is NULL. Therefore, when a list has only one item in it, then the `next` pointer of the last item is NULL and the `previous` points back to itself. In other words, if *head* is non-null, *head*$-$ >*previous* will always be the TAIL of the list.

Macros for manipulating lists of these structures are in the `list_macros.h` include file. This file is included by `pfs.h`, but may be included again without problems.

APPEND_ITEM(*new, head*) appends the item *new* to a doubly linked list pointed to by *head*. The item *new* goes at the TAIL of the list. It must not already be present in the list. This macro modifies its second argument, *head*, which should be a variable.

EXTRACT_ITEM(*item, head*) removes *item* from a doubly-linked list headed by *head*. If *item* is not a member of the list, the results are not defined. The extracted *item* will NOT be freed.

APPEND_LISTS(*list1,list2*) performs an O(1) list appending operation. It sets *list1* to point to a doubly-linked list consisting of *list1* appended to *list2*. *list2* must not already be a part of *list1* or the results are unpredictable. *list2* will be a garbage value at the end of this exercise, since it will no longer point to a valid doubly-linked list. *list1* and *list2* must already be valid doubly-linked lists.

3.6.2 Application-reserved (`app`) member of Prospero structures

This feature is available in prospero versions Alpha.5.3 and later.

For all exported structures in Prospero visible at the library interface (except for TOKENs), there is a special member named `app`, for applications writers. This member is a union:

```
union app {
int flg; /* Flag or number */
void *ptr;              /* Generic Pointer */
};
```

Each allocation/freeing package (eg: vlalloc(), vlfree(), vllfree(), vlcopy()) has a 'application freeing function specifying function' (e.g., vlappfree()) associated with it.

So, if we use the app.ptr member of the VLINK structure to point to a 'struct vlink_aux', to be freed (if non-null) with the 'vlauxlfree function, then code using the VLINK in that way will have to call: vlappfree(vlauxlfree); and then all subsequent calls to vlfree() will call vlauxlfree on the app.ptr member.

3.7 Entry Points

Entry points: **atalloc, atfree, atlfree, vlalloc, vlfree, vllfree, add_vlink, del_vlink, p_get_dir, mk_vdir, pget_am, pget_at, rd_vdir, rd_vlink, pfs_open,** and **pfs_fopen**.

PATTRIB atalloc(*void*), **FILTER flalloc**(*void*), **TOKEN tkalloc**(*char *s*), **ACL acalloc**(*void*), and **VLINK vlalloc**(*void*) allocate and initialize structures for storing attributes, filters, tokens, access control list entries, and virtual links. They call out_of_memory() on failure, which is a macro in `pfs.h` which currently raises an error condition and aborts program execution. Its behavior may be changed by resetting the value of the global variable **internal_error_handler** (defined in `pfs.h` to a function with some alternative behavior (such as popping up a window with a failure message and offering to restart the application or exit).[1] Since the only failure condition for these functions is running out of available memory, they do not set **perrno**.

atfree(*PATTRIB at*), **flfree**(*FILTER fl*), **acfree(***ACL ac***)**, and **vlfree**(*VLINK vl*) free the storage allocated to *at*, *fl*, *ac*, and *vl*. They also free any standard Prospero memory structures referenced by the members of these structures; for example, freeing a *VLINK* will also free any Prospero string referenced by the *VLINK*'s `host` member. They also free any memory referred to by the structure's `app.ptr` member, using whatever function the user set via **atappfree(), flappfree(), acappfree(),** or **vlappfree(),** as appropriate. The user can disable this feature, if it has already been enabled, by calling the appropriate *xx***appfree()** function with a null pointer as its argument.

atlfree(*at*) and **vllfree**(*vl*) free *at* and *vl* and all successors in a linked list of such structures. They do not return error codes nor do they set `perrno`, since they cannot fail. **tkalloc(***s***)** initializes the `token` member of the *TOKEN* structure it allocates to be a copy of *s*.

FILTER **flcopy(***FILTER fl, int recurse***)** will return a copy of the filter *fl*. If the *recurse* flag is non-zero, then it will also recursively copy the linked list of filters that *fl* points to the head of.

*char ** **stcopy(***char *s***)** allocates an area of memory large enough to hold the string *s* and copies *s* into it. It is usually used to store a string. The number of bytes allocated to a string can be checked with the macro **stsize(***char * string***)**. An alternative interface to **stcopy** is *char ** **stalloc(***size_t nbytes***)**. **stalloc** allocates an area of uninitialized memory large enough to hold *nbytes* bytes of data and returns a pointer to it. Another interface is *char ****stcopyr(***char *source, char *dest***)**. The sequence:

```
a = stcopyr("string", a);
```

will yield results functionally equivalent to the sequence:

```
stfree(a);
a = stcopy("string");
```

The only difference is that **stcopyr()** attempts to reuse the already allocated space, if available. This avoids the overhead of extra calls to **malloc()** and **free()**, and is therefore frequently more efficient than the equivalent longer sequence of calls. The existing Prospero libraries and utilities make frequent use of **stcopyr()** for this purpose.

Also note that `a = stcopyr("foo", (char *) NULL)` is equivalent to `a = stcopy("foo");` Memory allocated by all of these interfaces should be freed with **stfree(***st***)**. **stfree(**`(char *) NULL`**)** is a guaranteed no-op. The various interfaces to **stcopy()** all call `out_of_memory() when appropriate.`

A frequent cause of problems when using memory allocation functions is freeing the same chunk of memory twice. One may optionally enable consistency checking code in the allocators and freeing functions by defining `ALLOCATOR_CONSISTENCY_CHECK` in `pfs.h`. This code has not yet

[1] The Prospero directory server takes advantage of this and rebinds **internal_error_handler()** to a function that logs a message to the server's log file and attempts to restart the server.

been finished for the **stalloc()** family, but works for all other allocators. If any double freeing is detected, **internal_error_handler()** will be called.

A programmer may also easily check for memory leaks by looking at the global variables *int* **acl_count, pattrib_count, filter_count, pauth_count, pfile_count, token_count, vlink_count, and rreq_count** to see how many of each of the corresponding structures have been allocated.[2][3]

add_vlink(*direct,lname,l,flags*) adds a new link *l* to the directory named *direct* with the new link name *lname*. *direct* is a string naming the directory that is to receive the link. If *flags* is AVL_UNION, then the link is added as a union link. **add_vlink** returns PSUCCESS (0) on success and an error code on failure.

This interface to this function will change in a later version of the library to be **p_add_nlink()**, with a corresponding **p_add_link** that takes a *VLINK* instead of a string for the *direct* argument.

del_vlink(*path,flags*) deletes the link named by *path*. At present, *flags* is unused. **del_vlink** returns PSUCCESS (0) on success and an error code on failure.

This interface to this function will change in a later version of the library to be **p_del_nlink()**, with a corresponding **p_del_link()** that takes a *VLINK* instead of a string for the *path* argument.

p_get_dir(*VLINK dlink,char *components,VDIR dir,int flags,TOKEN acomp*) contacts the Prospero server on host *dlink->host* to read the directory *dlink->hsoname*, resolving union links that are returned and applying *dlink->filters*, if set.

If *components* is a null pointer, all links in the directory are returned. If *components* is a non-null string, only those links with names matching the string. The string may be a wildcarded name containing the `*` and `?` characters; these have their conventional meanings. The string may also be a regular expression, enclosed between parentheses; in that case, all links matching the regular expression are returned.

p_get_dir() will always, in addition to any other links it might return, return any link whose literal name is the *components* string. This feature means that you do not have to worry about retrieving links whose names contain special characters, even if more special characters are defined at some future time. An example: The *components* string `(ba`*na*`*na)`, in addition to matching `banana` and `bananana`, also (as an important special case) matches the component whose literal name is `(ba`*na*`*na)`.

dir is a Prospero directory structure that is filled in. *flags* can suppress the expansion of union links (GVD_UNION), force their expansion (GVD_EXPAND), request the return of link attributes on the *VLINK* structure's `lattrib` member (GVD_ATTRIB), and suppress sorting of the directory links (GVD_NOSORT). *acomp* should normally be NULL. For many applications, one does not need to call this procedure, and should use **rd_vdir** and **rd_vlink** instead. **p_get_dir** returns PSUCCESS (0) on success and an error code on failure.

The standard way to retrieve the attributes of a link in a directory is to call **p_get_dir** with the *dlink* argument pointing to the directory in which the link is located and the *components* argument being the name of the link whose attributes are to be retrieved.

Compatability note: **p_get_dir** was named **get_vdir** or **p_get_vdir()** in releases of Prospero before Alpha.5.2. Those older interfaces are still available but should be converted. release is backwards-compatible with those older uses. Please note that the Alpha.5.3 release of **p_get_dir()** will not moify the *dlink*, but that previous releases did corrupt the *dlink* while expanding a union lnk.

[2]We use this facility to debug the Prospero server; it returns this information in response to the `pstatus` command.

[3]Some of the structures mentioned in this list of global variables are not yet documented in this manual.

mk_vdir(*char path[], int flags*) creates a new virtual directory with the new name *path* in the currently active virtual system. *flags* should usually be 0; the only flag currently defined is MKVD_LPRIV, which causes the directory to be created with very limited permissions available to the creator. See the documentation of the `CREATE-OBJECT` command in the protocol specification if you want a better explanation of this option. **mk_vdir** returns PSUCCESS (0) on success and an error code on failure.

This interface to this function will change in a later version of the library to be **p_mk_ndir()**, with a corresponding **p_mk_dir** that takes a *VLINK* referring to the directory and a string which is the new link name.

pget_am(*VLINK link,TOKEN *ainfop, int methods*) returns the access method that should be used to access the object referenced by *link*. **ainfop* is a pointer to a variable of type TOKEN. When **pget_am** returns, this variable will be a NULL pointer if no appropriate access methods were available or will point to the value of the best ACCESS-METHOD attribute associated with the object referenced by *link* if appropriate methods were available. When more than one appropriate access method is available, **pget_am** attempts to choose the least expensive one.

methods is a bit-vector identifying the methods that are acceptable to the application. The methods presently supported are: the local filesystem (P_AM_LOCAL), anonymous FTP (P_AM_AFTP), regular FTP (P_AM_FTP), Sun's Network File System (P_AM_NFS), the Andrew File System (P_AM_AFS), the Gopher distributed directory service binary and text file retrieval protocols (P_AM_GOPHER), and telnettable services (P_AM_TELNET). Note that to effectively use the (P_AM_FTP) access method, the server on the remote end will have to know that the user has an account valid for FTP on the server. **pget_am** returns P_AM_ERROR (0) on failure and leaves an error code in `perrno`. Upon success, **pget_am** returns the value of the access method that was chosen.

This interface returns information that allows you to retrieve a file, but does not do any of the work of retrieving it. We expect most programmers to use the **pfs_open** or **pfs_fopen** interfaces instead. The only exception is the TELNET access methods

PATTRIB pget_at(*VLINK link,char atname[]*) returns a list of values of the *atname* attribute for the object referenced by *link*. If *atname* is NULL, all attributes for the referenced object are returned. If *atname* is a string, it is a string which is just a plus-separated list of attribute specification options to the `EDIT-OBJECT-INFO` protocol message. **pget_at** returns NULL on failure, or when no attributes are found. On failure, an error code is left in `perrno`. On success, `perrno` is explicitly set to `PSUCCESS`.

If the object has been forwarded, pget_at() will follow the forwarding pointers, just as other PFS library functions do. If the object has been forwarded, pget_at() will modify *link* so that the link's `host` and `hsoname` members refer to the link's new location.

This function will be renamed in a later version of this library. The new function will be named **p_get_at**.

rd_vdir(*dirarg,comparg,dir,flags*) lists the directory named by *dirarg* (relative to the current working directory or the root of the active virtual system) returning the links whose names match *comparg*. *dir* is a Prospero directory structure that is filled in. *flags* can suppress the expansion of union links (RVD_UNION), force their expansion (RVD_EXPAND), request the return of link attributes (RVD_ATTRIB), suppress sorting of the directory links (RVD_NOSORT), suppress use of cached data when resolving names (RVD_NOCACHE), or request the return of a reference to the named directory, suppressing the return of its contents (RVD_DFILE_ONLY). **rd_vdir** returns PSUCCESS (0) on success and an error code on failure.

As a special case, if the *comparg* is a null pointer or the empty string and the *dirarg* refers to a link that is not a DIRECTORY, then a directory entry containing a single link to the *vlink* named by comparg is returned; in that special case, this interface behaves similarly to **rd_slink**.

This function's interface will change; it will probably be renamed **p_get_ndir()**.

VLINK rd_vlink(*path*) is an alternative interface for resolving names. **rd_vlink** returns the single link named by *path*. Its function is equivalent to calling **rd_vdir** with *comparg* set to the last component of the path and *dirarg* set to the prefix. **rd_vlink** returns NULL on failure leaving an error code in `perrno`. **rd_vlink()** will also expand symbolic links it encounters, whereas **rd_vdir()** returns the symbolic links in a directory unexpanded.

VLINK **rd_slink**(*path*) works just like **rd_vlink**, except it will not expand symbolic links.

pfs_open(*VLINK vl,int flags*) and **FILE *pfs_fopen**(*VLINK vl, char *type*) are identical to **open** and **fopen** in the C library except that instead of a filename, they take a pointer to a Prospero virtual link structure and open the file referenced by the link. Note that they currently do not work to create files; indeed, they inherently can't, since they accept a pointer to an already existing link. **pfs_open** does not take the third optional *mode* argument that **open** takes, since Prospero's access control list mechanism does map well onto the UNIX protection modes.

For files which are not already mapped into the local UNIX filesystem, these functions work by retrieving the file as a temporary file; a reference to this temporary file is then returned. In the current implementation, we do not cache files; a new copy is retrieved every time you call pfs_open() or pfs_fopen(). If you want to use the same data more than once (e.g., display it via a paging program and then offer to save it), it will speed up your program substantially if you know that **pfs_open()** and **pfs_fopen()** return file references which you can run **lseek()** or **fseek()** on, respectively.

Until Prospero release Alpha.5.2, the **pfs_open** and **pfs_fopen** calls were in libpcompat, not in libpfs.

4 Pcompat Library

The compatibility library includes replacements for existing system calls and library routines that interact with the directory service. The replacements optionally resolve names using the Prospero file system. The behavior depends on the value of the `pfs_enable` global variable. Possible values are defined in *pcompat.h* and are described below. Note that `pfs_enable` exists in only a single instance, even in a multithreaded context; setting it for one thread sets it for all.

The default Prospero installation procedure leaves the compatibility library in `lib/pcompat/libpcompat.a`. Programs linked with the compatibility library should also be linked with the pfs library, since the compatibility library uses some functions in libpfs.

The routines in the compatibility library are self-initializing; they will call p_initialize() and initialize all data structures internally. This makes it possible to relink existing programs with the compatibility library with less effort.

Like the POSIX readdir(), the Prospero readdir() is not multithread-safe because it uses internal data structures. We would gladly accept a contributed readdir_r(), but will not write it ourselves until a strong need appears.

As of this writing, the compatibility library does not run on as many machines as the pfs library does. Specifically, the compatibility library is known not to work on HP-UX and on AIX.

Table 1: Settings for the `pfs_enable` global variable

Value	Meaning
PMAP_DISABLE	Never resolve names within the virtual system
PMAP_ENABLE	Always resolve names within the virtual system
PMAP_COLON	Resolve names within the virtual system if they contain a :
PMAP_ATSIGN_NF	Resolve names within the virtual system by default, but treat names beginning with an @ or full path names that don't exist in the virtual system as native file names
PMAP_ATSIGN	Resolve names within the virtual system by default, but treat names beginning with an @ as native file names

Therefore, use of the pfs library is suggested for maximal portability. The compatibility library is not compiled by default. (See the Prospero installation instructions for instructions on how to compile it.).

If you are interested in the pcompat library, you should also look at the Prospero User's Manual, in the section titled "The Compatibility Library".

Setting the environment variable `PFS_DEBUG` to a numeric value will cause the PCOMPAT library to set the PFS library's value of `pfs_debug` to that value. This is useful for debugging applications that use the compatibility library.

By default, the pfs compatability library will report prospero errors encountered by compatibility functions such as open() to `stderr`. You can silence these error messages by setting the variable `int pfs_quiet` in your program to a non-zero value or by defining your own global variable `pfs_quiet` with a non-zero initialized value. The global variable `pfs_quiet` is not multi-threaded; a single instance of it is shared among multi-threaded programs.

4.1 Entry Points

Entry points: closedir creat, execve, open, opendir, readdir, scandir, seekdir, stat, telldir, and pfs_access.

closedir, creat, execve, open, opendir, readdir, scandir, seekdir, stat, and **telldir** are identical to the entry points with the same names in the standard C library except that, depending on the value of the `pfs_enable` variable, file names may be resolved using Prospero.

pfs_access(*char *path,char *npath,int npathlen, int flags*) accepts a name, *path*, that is to be resolved using Prospero. **pfs_access** resolves the name, selects an access method, mounts the appropriate file system or retrieves the file if necessary, and returns a new name in *npath* that may be passed to open. *npath* must be a buffer large enough to hold the new name, and its size must be passed in *npathlen*. By setting *flags*, it is possible to specify that the file is to be created if it does not exist (PFA_CREATE), or to indicate that the file will be opened read only (PFA_RO). **pfs_access** returns PSUCCESS (0) or PMC_DELETE_ON_CLOSE on success. A return value of PMC_DELETE_ON_CLOSE indicates that the file has been cached on the local system and that the calling application should delete the cached copy when done with it. Any other return code indicates failure.

Warning: As of this writing, the PFA_CREATE flag has not been fully implemented.