M-270 Rev. 3

PDBLib User's Manual

for
PDBLib Version 10.0

Stewart A. Brown
Dennis Braddy
Janice E. Moura

Lawrence Livermore National Laboratory

Part of this work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Table of Contents

1.0 Introduction

PDBLib is a small library of file management routines useful for storing and retrieving binary data in a portable format. It is intended to supply a flexible means of managing binary databases without the user having to be concerned with the machines on which they are written or read.

The specific goals for this library are:

- The data files are portable. Using only simple binary file transfers, these files may be passed to any machine supporting an implementation of this library and the data in them immediately read.
- The library routines are portable. The library is written in C which is the most standardized language which also enjoys a broad range of implementations. In fact, a C compiler is probably the easiest piece of software to find for any given machine.
- The library will support call-by-reference or call-by-value interfaces. In this way the library can be used from FORTRAN (a call-by-reference language) or C (a call-by-value language) programs.
- 'Arbitrary' data structures may be defined to the PDB system so that there is maximum flexibility and ease of use for application programming. C structs or FORTRAN common blocks may be written or read as a whole. Record structured variables are thus supported.
- Data files may be generated for a particular machine type. By default when files are created they are written with the format of the machine that does so. However, they can be written in the format of a designated machine. Thus a more powerful machine can prepare files for a less powerful one.
- The library will be small so that machines with as little memory as about 512 Kbytes will be able to profitably use the PDB system.

The binary data format of the files is not machine independent, but it is portable. For the sake of efficiency, the data in any file is expressed in terms of some data format, e.g. IEEE format with 32 bit integers and floats. When PDB files are passed around on machines with the same data format, the data reads and writes involve no conversions. Only when one of these files is read on a machine with a different data format are conversions performed on the data it contains.

This document is divided into several sections. The next section gives an overview of the use of PDBLib and describes some programming practices which are necessary to the successful use of the library. It is followed by a section summarizing the C interface routines in PDBLib and a section giving a more formal and complete description of the C interface with information about normal and error return values, functionality, and rationale. Next are similar sections for the FORTRAN interface routines. Subsequent sections discuss PDBLib design philosophy, data structures, examples, and related documentation.

2.0 Overview of PDBLib

The functional PDBLib interface was designed as a minimal extension of the standard C library binary I/O interface. The relevant C library functions are **fopen**, **fclose**, **fwrite**, and **fread**. The extensions are driven mainly by some missing features in C. Because C doesn't handle data types as types, PDBLib must be told the type of a variable when it is written to a file. Because C variables don't know about their dimensions, PDBLib must be given the dimensions when a variable is written out. On the other hand, because PDBLib knows both type and dimension information about file variables, PDBLib requires less information from the programmer when reads are done.

Since C doesn't have the needed type handling facilities, data structures must be described to PDBLib. Therefore, in addition to the basic I/O functions mentioned above, there is a function in PDBLib to be used when a C data structure or a FORTRAN common block must be described in a file.

Next, to give the user the most flexibility and efficiency regarding data conversions, PDBLib has a function which lets the user specify the binary data format of the file. In this way, a file can be targeted for a particular machine type. A common use for this might be when a powerful machine produces some data which is going to be reprocessed on a less powerful machine. The more powerful machine which writes the file might target the file for the less powerful one so that the conversions are done by the faster machine and not the slower one.

Finally, in some applications with structures, some members of a structure may be pointers, and the actual type to which a pointer points may be changed by a cast depending on some other member's value. PDBLib supports the notion of a 'cast' in that it can be told to ignore the type of a structure member and get the actual type from another member when it writes out the structure.

Because all of the information about files and their contents is contained in PDBLib structures and hash tables, the C API (Application Program Interface) has no inquiry functions. Users can extract information directly from the **PDBfile** structure and they can use the hash table lookup function, **SC_def_lookup**, to get access to the **syment** and **defstr** structures which describe variables and types, respectively.

The FORTRAN API to PDBLib has many of the same functions as the C API plus several more inquiry functions. These functions are necessary because FORTRAN cannot easily work with structures.

There are also routines to manage data attributes. An attribute table will be created for a PDB file the first time **PD_def_attribute** is called. The attribute table is kept in the **PDBfile** structure. Variables can be added to PDB files solely for their attribute values. Attributes can be given to entire variables or any part of them. This works because the attribute table is a separate hash table, like the symbol table, which associates names with attribute values. It also is used to manage the attributes themselves. It is not exactly clear why one should use these attributes. Most of the utility of attributes is already inherent in the structures or records which can by built with PDBLib's mechanism for defining derived types. Since the structure handling mechanisms are much more efficient, the reason for using attributes instead of structures should be very clear in the mind of the application programmer.

2.1 Compiling and Loading

To compile your C programs you must use

```
#include <pdb.h>
```

in the source files which deal with the interpreter.

To link your application you must use the following libraries in the order specified.

-lpdb -lpml -lscore [-lm ...]

Although this is expressed as if for a UNIX linker, the order would be the same for any system with a single pass linker. The items in [] are optional or system dependent.

Each system has different naming conventions for its libraries and the reader is assumed to understand the appropriate naming conventions as well as knowing how to tell the linker to find the installed PACT libraries on each system that they use.

2.2 Memory Management Considerations

PDBLib is capable of correctly handling data that is indirectly referenced through arbitrary levels of pointers. In order to do this it is necessary to put an extra layer of memory management over the standard C library routines. The basic requirement is that given a pointer, one would like to know how many bytes of data it points to. The functions, **SC_alloc**, **SC_realloc**, **SC_strsave**, **SC_free**, and **SC_arrlen**, built on top of the standard C library functions, **malloc** and **free**, provide this capability. For C programmers, macros are provided which offer a nice and intuitive way of using these functions (they also provide an abstraction barrier against the details of any memory management scheme). These functions and macros are documented in the SCORE User's Manual.

A brief discussion of the procedure for writing and reading indirectly referenced data follows. Although the discussion will use integers as an example, the ideas apply to all data types, primitive or derived.

Consider the following:

```
int a[10], *b;
b = MAKE_N(int, 10);
```

Both *a* and *b* are pointers to 10 integers (macro **MAKE_N** is used to allocate the necessary space). The difference as far as an application is concerned is that the space that *a* points to was set aside by the compiler at compile time (for all practical purposes) while the space the *b* points to is created at run time. There is no possibility, given the definition of the C language, of asking the pointer *a* how many bytes it points to. On the other hand, since *b* is dynamically allocated, an extra layer of memory management could be provided so that a function, **SC_arrlen**, could be defined to return the number of bytes that a dynamically allocated space has. In particular,

SC_arrlen(a) => -1 (indicating an error)

SC_arrlen(b) => 10

SC_arrlen(b+5) => -1 (indicating an error)

These functions and macros can be found in SCORE. The general utility of something like **SC_arrlen** made it desirable to put it in the lowest possible level library. This could be used for example, to implement some dynamic array bound checking.

PDBLib uses this idea to be able to trace down arbitrary layers of indirection and obtain at each level the exact number of bytes to write out to a file. Of course, it also writes this information out so that the correct amount of space can be allocated by read operations as well as re-creating the correct connectivity of data structures.

Great care must be taken that pointers to fixed arrays not be imbedded in a chain of indirects unless their dimension specifications are included either in the I/O request or the definition of a structure. This point cannot be over-emphasized! The extra memory management layer may fail to detect a statically allocated array and return an erroneous byte count. This in turn will cause very obscure incorrect behavior (in the worst of circumstances) or a direct crash (the best outcome possible).

Also, note that subsets of dynamically allocated arrays cannot know how many bytes they contain and hence care should be taken in their use.

The example on the following page shows the different ways that statically allocated arrays, dynamically allocated arrays, statically allocated arrays of pointers, and dynamically allocated arrays of pointers are handled by PDBLib. Note: The function **SC_strsave** invokes the **MAKE_N** macro.

```
/* define variables in pairs - one to write and one to read into */
    PDBfile *strm;
    char *a, *b;
    char c[10], d[10];
    char *e[3], *f[3];
    char **s, **t;

    s = MAKE_N(char *, 2);

/* fill statically and dynamically allocated arrays */
    strcpy(c, "bar");
    a = SC_strsave("foo");

/* fill statically and dynamically allocated arrays of pointers */
    e[0] = SC_strsave("Foo");
    e[1] = NULL;
    e[2] = SC_strsave("Bar");
    s[0] = SC_strsave("Hello");
    s[1] = SC_strsave("world");

/* write these variables out
 * note the dimension specifications and the type
 */
    PD_write(strm, "c(10)", "char", c);
    PD_write(strm, "a", "char *", &a);
    PD_write(strm, "e(3)", "char *", e);
    PD_write(strm, "s", "char **", &s);

/* read the file variables into fresh memory spaces
 * note that the pointers to the pointers are passed in since the
 * space is to be allocated and the value of the pointer here must
 * be set to point to the new space
 */
    PD_read(strm, "c", d);
    PD_read(strm, "a", &b);
    PD_read(strm, "e", f);
    PD_read(strm, "s", &t);
```

2.3 PDB Syntax

This section contains discussion of some PDB concepts that are used elsewhere in this manual.

In the following discussion and elsewhere in the manual a BNF style notation is used. In such places the following constructs are used:

|	or
[]	items between the brackets are optional
[]*	zero or more instances of the items between the brackets
[]+	one or more instances of the items between the brackets
“x”	literal x

Since a data entry can be of a structured type, in general, a data entry is a tree. A part of the data tree is specified by a path to that part. A path specification is of the form:

path	:= node_spec | path.node_spec
node_spec	:= name[“[“ index_expr “]”]* | name[“(“ index_expr “)”]*
name	:= entry-name | member-name

The last node specified is referred to as the terminal node.

2.4 PDBLib Rules

There are a few crucial rules to be followed with PDBLib. If they are not obeyed, PDBLib may crash or the results desired will not occur.

1. The following reserved characters are not allowed in variable names or **defstr** member names: ‘.’, ‘(‘, ‘)’, ‘[‘, and ‘]’. They are reserved by the grammar of the data description language which follows some C and some FORTRAN conventions.

2. In each read and write operation, the type of the argument corresponding to the variable to be written or read must be a pointer to an object of the type specified for the entry. For example,

```
    int *a, *b;
            .
            .
    PD_write(strm, "a", "integer *", &a);
    PD_read(strm, "a", &b);
            .
            .
```

3. When using pointers and dynamically allocated memory with PDBLib, use **SC_alloc**, **SC_realloc**, **SC_strsave**, **MAKE**, **MAKE_N**, **REMAKE**, or **REMAKE_N** to allocate memory. These functions and macros are documented in the SCORE User’s Manual.

4. When reading or writing part of a variable, especially a structured variable, the terminal node must be of primitive type or a structure containing no indirections and whose descendant members contain no indirections. Furthermore, the path to the desired part must contain one array reference for each level of indirection traversed. For example,

```
    char **s, c;

/* fill s so that s[0] = "Hello" and s[1] = "World" */
            .
            .
    PD_write(strm, "s", "char **", &s);

/* read the 'o' in "Hello" */
    PD_read(strm, "s[0][4]", &c);

/* read the 'o' in "World" */
    PD_read(strm, "s[1][1]", &c);
            .
            .
```

3.0 Summary of the C API

There is a hierarchy of routines in PDBLib from high to low level. The high level routines form the API while the lower level routines are modularized to perform the actual work. It should be noted that the lower level routines are sufficiently well modularized so as to make it possible to build entirely different API's for PDBLib.

The high level PDBLib routines have a strict naming convention. All high level routines begin with 'PD_'. Some routines have several related forms. These are referred to as families of functions. For example the PD_write family.

Most of these routines put an error message into a global variable called **PD_err**. The error messages include the name of the routine in which they are made thus eliminating the need for error codes which must be cross referenced with some other document. In this way application programs can check for error conditions themselves and decide in what manner to use the PDBLib error messages instead of having error messages printed by the system routines. Error messages are not stacked and must be processed by the application before any other PDBLib calls are made in order to avoid potential overwrites. See the descriptions of individual routines for more information about error handling and messages.

Programs written in C must include a header which makes certain declarations and definitions needed to use the library. Much in the same spirit as one includes **stdio.h** to use **printf** and others, include **pdb.h** as follows:

```
#include "pdb.h"
```

The file **pdb.h** #include's some other files which must either be in your directory space or pointed to in some manner which your compiler can recognize. The auxiliary #include files are **schash.c**, **scstd.h**, and **score.h**. These files are a part of the SCORE package which you must have to use PDBLib.

3.1 Some Guidance in Using the C API to PDBLib

PDBLib offers a great deal of flexibility to its users. In some cases the flexibility results in a bewildering assortment of function calls from which the application developer must choose. This section is intended to offer some guidance and enable developers to home in on the functions they need to use.

3.1.1 Opening, Creating and Closing PDB Files

These are the most fundamental operations involving PDBLib. The function **PD_open** is used to either open an existing file or create a new one. **PD_close** is used to close a PDB file so that it can be recognized by PDBLib for future operations.

PDBLib allows applications to specify the binary format in which a newly created file will be written. **PD_target** does this work. It is not necessary to invoke **PD_target** before creating a new PDB file. In this case the binary format is that appropriate for the host system.

During the run of an application code, PDBLib can complete the information in an open PDB file so that in the event of a code crash, the file will be a valid PDB file. This functionality is provided by **PD_flush**.

3.1.2 Writing Data to PDB Files

The following paragraphs describe a sequence of increasingly more elaborate output operations for PDB files. They are all compatible with one another so users can select the ones which match their needs best. The most straightforward operations are first.

There are two forms for the most basic data writing operations. These have to do with how the application wants to handle the dimension specifications. The two functions are **PD_write** and **PD_write_alt**.

PDBLib supports the notion of writing data of one type out into a file as another type. More precisely, an integer type of data can be written to a file in the format of any other integer type, and similarly for floating point types. The application must take all responsibility for ensuring the appropriateness of this type of conversion (e.g. underflows and overflows). The functions which support this are **PD_write_as** and **PD_write_as_alt**.

PDBLib allows applications to append data to existing entries. This is handy in situations where the total amount of data is not known in advance, but a logical ordering of the data is apparent which matches the order in which data will be written. The functions which do this are the **PD_append** family of functions.

Finally, PDBLib allows applications to reserve space on disk and then let subsequent writes fill in that space with values. This is handy in instances where an application knows a logical structure for a data set but needs to write it out in smaller pieces and not necessarily in the order implied by its logical structure. The functions which let applications reserve space are **PD_defent** and **PD_defent_alt**. Reserved spaces may be written to with any of the **PD_write** family of commands.

3.1.3 Reading Data from PDB Files

Since data in a file has a definite size and shape, the reading operations in PDBLib are somewhat simpler than the writing operation. The **PD_read** family of operations which is the counterpart of the **PD_write** family does all of the work in reading data from files.

3.1.4 Defining New Data Types

To aid application developers in using structured data PDBLib lets applications define new data types in a PDBfile. New data types may either be derived from other existing data types or they may be primitive (integer types, floating point types, or byte sequences). To define derived types applications should use **PD_defstr** or **PD_defstr_alt**. To define primitive types use **PD_defix**, **PD_defloat**, or **PD_defncv**.

Since PDBLib supports pointered data types, it is often the case that a pointer in a derived type may point to data of any kind. In C, casts are used to control this behavior. PDBLib permits a member of a struct which is of type “char *” specify the actual type of another pointered member. The function **PD_cast** is used to set up this behavior.

3.1.5 Defining Attributes

PDBLib supports a general mechanism for managing a class of data which is variously referred to as attributes or meta data. In a great many cases, the careful design of data structures obviates the need for this kind of data. Nevertheless, PDBLib supplies four functions to manage attributive data: **PD_def_attribute**, **PD_rem_attribute**, **PD_get_attribute**, **PD_set_attribute**.

3.1.6 Using Pointers

The ability to use pointers with PDBLib is very powerful. It is unfortunate perhaps that it can also be a little bit tricky. Application developers should be aware of a key issue when using pointers with PDBLib. In general, when you want to write a data tree, you want to faithfully reproduce the connectivity of the tree. That means that if many pointers refer to the same chunk of memory then the file will exhibit that same pattern. To do this, PDBLib remembers the pointers that it has written and if handed a pointer that it has already written it does NOT write that memory into the file again. It instead makes note of the connection to an already existing region of disk. These same considerations apply to reading with the requisite changes in sense.

The difference between files and memory is that data trees may be created, destroyed, and pointers recycled. For instance, an application may wish to allocate an array, write it as an indirect, change it contents in memory, and write it out again under another name. This is a very common pattern of usage. If PDBLib remembers the pointers, only the first write will put any data on disk. Now PDBLib has no way of knowing what the application wants. The control that applications have is to remove the list of pointers that PDBLib knows about. This accomplished with the **PD_reset_ptr_list** command.

3.2 Terms

A few definitions must be given first to keep the following summary concise. Some of these will be elaborated upon in the next section.

PDBfile: for the purposes of a program this is a collection of all the relevant information about one of these data files (in the actual coding it is a C struct). See the section on Data Structures for more information.

ASCII: a string

TRUE: a value of 1 (defined by PDBLib)

FALSE: a value of 0 (defined by PDBLib)

VOID: any C data type

LAST: a pointer to 0 (declared and defined by PDBLib)

ENTRY: for the purposes of a program this is a collection of all the relevant information about a variable written to a PDB file (in the actual coding it is a C struct). See the discussion of **syment** in the section on Data Structures.

DEFSTR: for the purposes of a program this is a collection of all the relevant information about a data type in a PDB file (in the actual coding it is a C struct). See the discussion of **defstr** in the section on Data Structures.

3.3 Function Summary

These routines form the interface between C application programs and the PDBLib system.

PD_append(file, name, var)

Append the data pointed to by *var* to an existing entry under *name* in PDB file *file*.

Input: *file*, a pointer to a **PDBfile**,

name, an ASCII string containing the name of the variable in the PDB file and any dimension information,

var, a pointer to the location where the data is stored in memory.

Output: TRUE if successful, and NULL otherwise.

PD_append_alt(file, name, var, nd, ind)

Append the data pointed to by *var* to an existing entry *name* in PDB file *file*. In this alternate to **PD_append** dimension information is provided in arguments *nd* and *ind*.

Input: *file*, a pointer to a **PDBfile**,

name, an ASCII string containing the name of the variable in the PDB file,

var, a pointer to the location where the data is stored in memory.

nd, an integer containing the number of dimensions,

ind, an array of long integers containing (min, max, stride) triples for each dimension.

Output: TRUE if successful, and NULL otherwise.

PD_append_as(file, name, intype, var)

Append the data pointed to by *var* with type *intype* to an existing entry *name* in PDB file *file*.

Input: *file*, a pointer to a **PDBfile**,

name, an ASCII string containing the name of the variable in the PDB file and any dimension information,

intype, an ASCII string containing the type of the data pointed to by *var*,

var, a pointer to the location where the data is stored in memory.

Output: TRUE if successful, and NULL otherwise.

PD_append_as_alt(file, name, intype, var, nd, ind)

Append the data pointed to by *var* with type *intype* to the entry *name* in PDB file *file*. In this alternate to **PD_append_as** dimension information is provided in arguments *nd* and *ind*.

Input: *file*, a pointer to a **PDBfile**,

name, an ASCII string containing the name of the variable in the PDB file,

intype, an ASCII string containing the type of the data pointed to by *var*,

var, a pointer to the location where the data is stored in memory.

nd, an integer containing the number of dimensions,

ind, an array of long integers containing (min, max, stride) triples for each dimension.

Output: TRUE if successful, and NULL otherwise.

PD_cast(file, type, memb, contr)

This function tells PDBLib to substitute the type specified by the ASCII string in the member *contr* for the type of member, *memb*, in the structure, *type*.

Input: *file,* a pointer to a **PDBfile**,

type, an ASCII string containing the name of the data type in the PDB file,

memb, an ASCII string containing the name of the member of *type* whose type is to be replaced,

contr, an ASCII string containing the name of the member of *type* whose ASCII string value is to replace the type of the member, *memb*.

Output: TRUE if successful, FALSE otherwise.

PD_cd(file, dirname)

Change the current directory in file *file* to *dirname*.

Input: *file,* a pointer to a **PDBfile**,

dirname, an ASCII string containing the path name of the directory to change to.

Output: TRUE if successful, FALSE otherwise.

PD_close(file)

Close the file *file*. This function is used to complete and close a PDB file.

Input: *file*, a pointer to a **PDBfile**.

Output: TRUE if successful, FALSE otherwise.

PD_def_attribute(file, attr, type)

Define an attribute named *attr* of type *type* in PDB file *file*.

Input: *file*, a pointer to a **PDBfile**,

attr, an ASCII string containing the name of the attribute to be added to the PDB file attribute table,

type, an ASCII string containing the type of the attribute.

Output: TRUE if successful, FALSE otherwise.

PD_defent(file, name, outtype)

Define an entry in the symbol table of the PDB file *file*. This function reserves space on disk but writes no data. The data can be written with later PDBLib calls.

Input: *file*, a pointer to a **PDBfile**,

name, an ASCII string containing the name of the variable and any dimensional information,

outtype, an ASCII string specifying the type of data in the file.

Output: a pointer to the **syment** for the entry, if successful, and NULL otherwise.

PD_defent_alt(file, name, outtype, nd, ind)

Define an entry in the symbol table of the PDB file *file*. This function reserves space on disk but writes no data. The data can be written with later calls.

Input: *file*, a pointer to a **PDBfile**,

name, an ASCII string containing the name of the variable (no dimensional information can be supplied with the name),

outtype, an ASCII string specifying the type of data in the file,

nd, the number of dimensions of the entry,

find, an array of long integers containing (min, max) index range pairs for each dimension.

Output: a pointer to the **syment** for the entry, if successful, and NULL otherwise.

PD_defix(file, name, bytespitem, align, flg)

Define a primitive integral type (fixed point type) in the PDB file *file*.

Input: *file*, a pointer to a **PDBfile**,

name, an ASCII string containing the name of the new data type,

bytespitem, the number of bytes required for 1 item of the new type,

align, the byte alignment for the type,

flg, a flag indicating whether the byte ordering of the type is normal or reverse ordered (**NORMAL_ORDER** or **REVERSE_ORDER**).

Output: a pointer to the **defstr** for the type, if successful, and NULL otherwise.

PD_defloat(file, name, bytespitem, align, ordr, expb, mantb, sbs, sbe, sbm, hmb, bias)

Define a new floating point type to the PDB file *file*.

Input: *file*, a pointer to a **PDBfile**,

name, an ASCII string containing the name of the data type in the PDB file,

bytespitem, the number of bytes required for an item of the new type,

align, the byte alignment for this type,

ordr, an array of *bytespitem* integers specifying the byte order,

expb, the number of exponent bits,

mantb, the number of mantissa bits,

sbs, the position of the sign bit,

sbe, the starting bit of the exponent,

sbm, the starting bit of the mantissa,

hmb, the value of the high order mantissa bit,

bias, the bias of the exponent.

Output: a pointer to the **defstr** for the type, if successful, and NULL otherwise.

PD_defncv(file, name, bytespitem, align)

Define a primitive type that will not undergo format conversion from platform to platform in the PDB file *file*.

Input: *file*, a pointer to a **PDBfile**,

name, an ASCII string containing the name of the new data type,

bytespitem, the number of bytes required for 1 item of the new type,

align, the byte alignment for the type.

Output: a pointer to the **defstr** for the type, if successful, and NULL otherwise.

PD_defstr(file, name, mem_1, ..., mem_n, LAST)

Define a data structure called *name* with members mem_1, ..., mem_n. **LAST** must terminate the list of members. This uses a C-like syntax for the member definition and *name* is used in subsequent writes to refer to this type of data structure.

Input: *file*, a pointer to a **PDBfile**,

name, an ASCII string containing the name of the data structure type in the PDB file,

mem_i, an ASCII string representing the declaration of a member of a data structure.

Output: a pointer to the **defstr** for the data structure, if successful, and NULL otherwise.

PD_defstr_alt(file, name, nmemb, memb)

Define a data structure called *name* with members specified by memb.. This uses a C-like syntax for the member definition and *name* is used in subsequent writes to refer to this type of data structure.

Input: *file*, a pointer to a **PDBfile**,

name, an ASCII string containing the name of the data structure type in the PDB file,

nmemb, an integer number of members in the data structure and the length of *memb*,

memb, an array of ASCII string representing the declarations of the members of a data structure.

Output: a pointer to the **defstr** for the data structure, if successful, and NULL otherwise.

PD_family(file, flag)

If the file, *file*, has exceeded its size limit, open and return a new member of the file family. If *flag* is TRUE close the old file.

Input: *file*, a pointer to a **PDBfile**.

flag, an integer flag.

Output: a pointer to a **PDBfile**.

PD_flush(file)

Flush the file *file*. This function is used to write the descriptive information of a PDB file to disk.

Input: *file*, a pointer to a **PDBfile**.

Output: TRUE if successful, FALSE otherwise.

PD_get_attribute(file, var, attr)

Get the value of the *attr* attribute of variable *var* in PDB file *file*. Variables only have values for attributes which are explicitly given.

Input: *file*, a pointer to a **PDBfile**,

var, an ASCII string containing the name of the variable in the PDB file,

attr, an ASCII string containing the name of the attribute in the PDB file attribute table.

Output: a pointer to the attribute value, if successful, and NULL otherwise.

PD_get_max_file_size(file)

Get the current maximum size for PDB file *file*.

Input: *file*, a pointer to a **PDBfile**.

Output: Return the integer value of the current maximum file size.

PD_get_mode(file)

Get the current mode of PDB file *file*: 2 (append - ’a’), 3 (open - ’r’), 4 (create - ’w’).

Input: *file*, a pointer to a **PDBfile**.

Output: Return the integer value of the current mode.

PD_get_offset(file)

Get the current default offset for PDB file *file*.

Input: *file*, a pointer to a **PDBfile**.

Output: Return the integer value of the default offset.

PD_ln(file, var, link)

Create a link to a variable in PDB file *file*.

Input: *file,* a pointer to a **PDBfile**,

var, an ASCII string containing the path name of the variable to link to,

link, an ASCII string containing the path name of the link.

Output: TRUE if successful, FALSE otherwise.

PD_ls(file, path, type, num)

Return a list of names of entries (variables and/or directories) in PDB file *file* that are of a type *type* and that are in the directory and match the variable name pattern specified by *path*.

Input: *file,* a pointer to a **PDBfile**,

path, an ASCII string containing the path name of the directory to search and/or the variable name pattern to match,

type, an ASCII string containing the type of variables to return,

num, the number of entries returned.

Output: a pointer to an array of pointers to entry names, if successful, and NULL otherwise.

PD_make_image(name, type, data, dx, dy, bpp, xmin, xmax, ymin, ymax, zmin, zmax)

Create and initialize a **PD_image** data structure.

Input: *name*, an ASCII string containing the name of the image,

type, an ASCII string containing the data type of the image elements (e.g. “char”, “float”, “complex”),

data, pointer to array of image data of type type,

dx, width of image,

dy, height of image,

bpp, bits per pixel of the image,

xmin, minimum value of coordinate associated with image x direction,

xmax, maximum value of coordinate associated with image x direction,

ymin, minimum value of coordinate associated with image y direction,

ymax, maximum value of coordinate associated with image y direction,

zmin, minimum value of image data (for palette labelling),

zmax, maximum value of image data (for palette labelling).

Output: a pointer to a **PD_image**.

PD_mkdir(file, dirname)

Create a directory *dirname* in PDB file *file*.

Input: *file,* a pointer to a **PDBfile**,

dirname, an ASCII string containing the new directory path name.

Output: TRUE if successful, FALSE otherwise.

PD_open(filename, mode)

Open a PDB file on disk named *filename* in one of three modes: “w” for create, “r” for read-only, or “a” for append. This function is used to create a new PDB file or to open an existing PDB file for reading or for adding new entries.

Input: *filename*, an ASCII string,

mode, an ASCII string, either “w” for create, “r” for read, or “a” for append.

Output: a pointer to a **PDBfile**, if successful, and NULL otherwise.

PD_put_image(file, image, index)

Write image *image* to file *file* with index *index*.

Input: *file*, a pointer to a **PDBfile**,

image, a pointer to the image data structure to be written,

index, an unique integer ordinal index.

Output: TRUE if successful, FALSE otherwise.

PD_put_mapping(file, mapping, index)

Write mapping *mapping* to file *file* with index *index*.

Input: *file*, a pointer to a **PDBfile**,

mapping, a pointer to the mapping data structure to be written,

index, an unique integer ordinal index.

Output: TRUE if successful, FALSE otherwise.

PD_put_set(file, set)

Write set *set* to file *file*.

Input: *file*, a pointer to a **PDBfile**,

set, a pointer to the set data structure to be written.

Output: TRUE if successful, FALSE otherwise.

PD_pwd(file)

Return the current directory in PDB file *file*.

Input: *file,* a pointer to a **PDBfile**.

Output a pointer to the path name of the current directory, if successful, and NULL otherwise.

PD_read(file, name, var)

Read a variable *name* from PDB file *file* and put it in *var*.

Input: *file*, a pointer to a **PDBfile**,

name, an ASCII string containing the name of the variable in the PDB file,

var, a pointer to the location where the data is to be placed.

Output: TRUE if successful, FALSE otherwise.

PD_read_alt(file, name, var, ind)

Read part of a variable *name* from PDB file *file* and put it in *var*.

Input: *file*, a pointer to a **PDBfile**,

name, an ASCII string containing the name of the variable in the PDB file,

var, a pointer to the location where the data is to be placed.

ind, an array of integer triples specifying a start, stop, and step index for each dimension of the entry.

Output: TRUE if successful, FALSE otherwise.

PD_read_as(file, name, type, var)

Read a variable *name* from PDB file *file* and put it in *var* as data of type *type*.

Input: *file*, a pointer to a **PDBfile**,

name, an ASCII string containing the name of the variable in the PDB file,

type, an ASCII string containing the type of the data desired,

var, a pointer to the location where the data is to be placed.

Output: TRUE if successful, FALSE otherwise.

PD_read_as_alt(file, name, type, var, ind)

Read part of a variable *name* from PDB file *file* and put it in *var* as data of type *type*.

Input: *file*, a pointer to a **PDBfile**,

name, an ASCII string containing the name of the variable in the PDB file,

type, an ASCII string containing the type of the data desired,

var, a pointer to the location where the data is to be placed.

ind, an array of integer triples specifying a start, stop, and step index for each dimension of the entry.

Output: TRUE if successful, FALSE otherwise.

PD_rel_image(image)

Release the space associated with a **PD_image** data structure.

Input: *image*, a pointer to a **PD_image**.

Output: None.

PD_rem_attribute(file, attr)

Remove an attribute named *attr* from the attribute table in PDB file *file*. All attribute values are removed simultaneously with the attribute.

Input: *file*, a pointer to a **PDBfile**,

attr, an ASCII string containing the name of the attribute in the PDB file attribute table.

Output: TRUE if successful, FALSE otherwise.

PD_reset_ptr_list(file)

Free the list of pointers which the PDB file *file* knows about. This includes both pointers in memory acquired during write operations and pointers in the file acquired during read operations.

Input: *file*, a pointer to a **PDBfile**,

Output: TRUE if successful, FALSE otherwise.

PD_set_attribute(file, var, attr, val)

Set the value of an attribute *attr* of the *var* in PDB file *file* to *val*. Variables can be added to PDB files solely for their attributes; that is, *var* need not be a variable in *file*.

Input: *file*, a pointer to a **PDBfile**,

var, an ASCII string containing the name of the variable in the PDB file,

attr, an ASCII string containing the name of the attribute in the PDB file attribute table,

val, a pointer to the value of the attribute *attr* to be associated with *var*.

Output: TRUE if successful, FALSE otherwise.

PD_set_max_file_size(file, v)

Set the maximum size for PDB file *file*.

Input: *file*, a pointer to a PDBfile.

v, an integer value for the maximum file size

Output: Return the new integer value of the maximum file size.

PD_set_offset(file, v)

Set the default offset for PDB file *file*.

Input: *file*, a pointer to a **PDBfile**,

v, an integer value for the default offset.

Output: Return the new integer value of the default offset.

PD_target(std, align)

Set up the next PDB file as if written by a machine of type defined by the given **data_standard** and **data_alignment** information. This function must be called before the **PD_open** call which creates the file.

Input: *std*, a pointer to a **data_standard** structure,

align, a pointer to a **data_alignment** structure.

Output: TRUE if successful, FALSE otherwise.

PD_typedef(file, oname, tname)

Define an alias for an existing type in the specified file.

Input: *file*, a pointer to a **PDBfile**,

oname, an ASCII string containing the name of the old data type,

tname, an ASCII string containing the name of the new data type.

Output: a pointer to the **defstr** for the type, if successful, and NULL otherwise.

PD_write(file, name, type, var)

Write the data pointed to by *var* under *name* and with *type* to PDB file *file*.

Input: *file*, a pointer to a **PDBfile**,

name, an ASCII string containing the name of the variable in the PDB file,

type, an ASCII string containing the type of the variable,

var, a pointer to the location where the data is stored in memory.

Output: TRUE if successful, and NULL otherwise.

PD_write_alt(file, name, type, var, nd, ind)

Write the data pointed to by *var* under *name* and with *type* to PDB file *file*. In this alternate to **PD_write** dimension information is provided in arguments *nd* and *ind*.

Input: *file*, a pointer to a **PDBfile**,

name, an ASCII string containing the name of the variable in the PDB file,

type, an ASCII string containing the type of the variable,

var, a pointer to the location where the data is stored in memory.

nd, an integer containing the number of dimensions,

ind, an array of long integers containing (min, max, stride) triples for each dimension.

Output: TRUE if successful, and NULL otherwise.

PD_write_as(file, name, intype, outtype, var)

Write the data pointed to by *var* under *name* and with *intype* to PDB file *file* as data of *outtype*.

Input: *file*, a pointer to a **PDBfile**,

name, an ASCII string containing the name of the variable in the PDB file,

intype, an ASCII string containing the type of the variable,

outtype, an ASCII string containing the output type of the variable,

var, a pointer to the location where the data is stored in memory.

Output: TRUE if successful, and NULL otherwise.

PD_write_as_alt(file, name, intype, outtype, var, nd, ind)

Write the data pointed to by *var* under *name* and with *intype* to PDB file *file* as data of *outtype*. In this alternate to **PD_write_as** dimension information is provided in arguments *nd* and *ind*.

Input: *file*, a pointer to a **PDBfile**,

name, an ASCII string containing the name of the variable in the PDB file,

intype, an ASCII string containing the type of the variable,

outtype, an ASCII string containing the output type of the variable,

var, a pointer to the location where the data is stored in memory.

nd, an integer containing the number of dimensions,

ind, an array of long integers containing (min, max, stride) triples for each dimension.

Output: TRUE if successful, and NULL otherwise.

4.0 The C API

In this section a more formal description of the C API for PDBLib is given including a more detailed account of their workings.

4.1 PD_APPEND

```
int PD_append(PDBfile *file
              char *name,
              void *vr)
```

Append data to an entry in the specified file. The type is assumed to be the same as for the original entry. The dimensions of the appended data are specified in *name*. They must match the original entry except for the most slowly varying one. The specification of the most slowly varying dimension must be one of the following:

min:max => new dimension is old_min:old_max+(max-min+1)

if min is the default_offset for the file

or

min:max => new dimension is old_min:max

if min is old_max+1

The rationale for this function is that some data sets are of unknown size until they are completely written. PDBLib permits any entry to reside in discontiguous blocks of disk space. The library is responsible for reading and writing data correctly across these blocks.

The shape or dimensional information of the entry is a part of the name string. In this respect **PD_append** behaves just like **PD_write**.

Input to this function is: *file*, a pointer to a **PDBfile**, *name*, an ASCII string containing the name of the variable and any dimensional information, and *vr*, a pointer to the data to be appended.

This function returns TRUE, if successful; otherwise, FALSE.

See also **PD_append_alt**, **PD_append_as**, **PD_append_as_alt**, **PD_defent**, **PD_defent_alt**, **PD_write**, **PD_write_alt**, **PD_write_as**, **PD_write_as_alt**.

```
int PD_append(PDBfile *file
              char *name,
              void *vr)
```

Example

```
#include “pdb.h”

    PDBfile *file;
    float *fv;

          .
          .
          .
    if (PD_append(file, “x(20)”, fv)) == FALSE)
       printf(“%s”, PD_err);
          .
          .
          .
```

Compare this with the example of **PD_write**.

4.2 PD_APPEND_ALT

```
int PD_append_alt(PDBfile *file
                  char *name,
                  void *vr,
                  int nd,
                  long *ind)
```

Append data to an entry in the specified file. The type is assumed to be the same as for the original entry. The dimensions of the appended data are specified in the *ind* array. They must match the original entry except for the most slowly varying one. The specification of the most slowly varying dimension must be one of the following;

min:max:stride => new dimension is old_min:old_max+(max-min+1)

if min is the default_offset for the file

or

min:max:stride => new dimension is old_min:max

if min is old_max+1

The rationale for this function is that some data sets are of unknown size until they are completely written. PDBLib permits any entry to reside in discontiguous blocks of disk space. The library is responsible for reading and writing data correctly across these blocks.

Input to this function is: *file*, a pointer to a **PDBfile**, *name*, an ASCII string containing the name of the variable; *vr*, a pointer to the data to be appended; *nd*, an integer number of dimensions; and *ind*, an array of longs with triples (start, stop, step) defining dimension information.

This function returns TRUE, if successful; otherwise, FALSE.

See also **PD_append**, **PD_append_as**, **PD_append_as_alt**, **PD_defent**, **PD_defent_alt**, **PD_write**, **PD_write_alt**, **PD_write_as**, **PD_write_as_alt**.

```
int PD_append_alt(PDBfile *file
                  char *name,
                  void *vr,
                  int nd,
                  long *ind)
```

Example

```
#include "pdb.h"

    PDBfile *file;
    float *fv;
    long ind[3]

          .
          .
          .
    ind[0] = 0L;
    ind[1] = 20L;
    ind[2] = 1L;
    if (PD_append_alt(file, "x", fv, 1, ind)) == FALSE)
       printf("%s", PD_err);
          .
          .
          .
```

Compare this with the example of **PD_write_alt**.

4.3 PD_APPEND_AS

```
int PD_append_as(PDBfile *file
                 char *name,
                 char *intype,
                 void *vr)
```

Append data to an entry in the specified file. The output type is assumed to be the same as for the original entry. The dimensions of the appended data are specified in the *name*. They must match the original entry except for the most slowly varying one. The specification of the most slowly varying dimension must be one of the following;

min:max => new dimension is old_min:old_max+(max-min+1)

if min is the default_offset for the file

or

min:max => new dimension is old_min:max

if min is old_max+1

The rationale for this function is that some data sets are of unknown size until they are completely written. PDBLib permits any entry to reside in discontiguous blocks of disk space. The library is responsible for reading and writing data correctly across these blocks.

The shape or dimensional information of the entry is a part of the name string. In this respect **PD_append_as** behaves just like **PD_write_as**.

Input to this function is: *file*, a pointer to a **PDBfile**, *name*, an ASCII string containing the name of the variable and any dimensional information, *intype*, an ASCII string specifying the type of the data to which *vr* points; and *vr*, a pointer to the data to be appended.

This function returns TRUE, if successful; otherwise, FALSE.

See also **PD_append**, **PD_append_alt**, **PD_append_as_alt**, **PD_defent**, **PD_defent_alt**, **PD_write**, **PD_write_alt**, **PD_write_as**, **PD_write_as_alt**.

```
int PD_append_as(PDBfile *file
                 char *name,
                 char *intype,
                 void *vr)
```

Example

```
#include “pdb.h”

    PDBfile *file;
    double *dv;

          .
          .
          .
    if (PD_append_as(file, “x(20)”, “float”, dv)) == FALSE)
       printf(“%s”, PD_err);
          .
          .
          .
```

Compare this with the example of **PD_write_as**.

4.4 PD_APPEND_AS_ALT

```
int PD_append_as_alt(PDBfile *file
                     char *name,
                     char *intype,
                     void *vr,
                     int nd,
                     long *ind)
```

Append data to an entry in the specified file. The output type is assumed to be the same as for the original entry. The dimensions of the appended data are specified in the *ind* array. They must match the original entry except for the most slowly varying one. The specification of the most slowly varying dimension must be one of the following;

min:max:stride => new dimension is old_min:old_max+(max-min+1)

if min is the default_offset for the file

or

min:max:stride => new dimension is old_min:max

if min is old_max+1

The rationale for this function is that some data sets are of unknown size until they are completely written. PDBLib permits any entry to reside in discontiguous blocks of disk space. The library is responsible for reading and writing data correctly across these blocks.

Input to this function is: *file*, a pointer to a **PDBfile**, *name*, an ASCII string containing the name of the variable; *intype*, an ASCII string specifying the type of data to which *vr* points; *vr*, a pointer to the data to be appended; *nd*, an integer number of dimensions; and *ind*, an array of longs with triples (start, stop, step) defining dimension information.

This function returns TRUE, if successful; otherwise, FALSE.

See also **PD_append**, **PD_append_alt**, **PD_append_as**, **PD_defent**, **PD_defent_alt**, **PD_write**, **PD_write_alt**, **PD_write_as**, **PD_write_as_alt**.

```
int PD_append_as_alt(PDBfile *file
                     char *name,
                     char *intype,
                     void *vr,
                     int nd,
                     long *ind)
```

Example

```
#include “pdb.h”

    PDBfile *file;
    float *fv;
    long ind[3]

          .
          .
          .
    ind[0] = 0L;
    ind[1] = 20L;
    ind[2] = 1L;
    if (PD_append_as_alt(file, “x”, “double”, fv, 1, ind)) == FALSE)
       printf(“%s”, PD_err);
          .
          .
          .
```

Compare this with the example of **PD_write_as_alt**.

4.5 PD_CAST

```
int PD_cast(PDBfile *file,
            char *type,
            char *memb,
            char *contr)
```

Dynamically change the type of a structure member. PDBLib supports an extended data typing mechanism called a structure. A structure is a set of declarations of members. Each member is in turn a data type known to the system. In some applications, a structure member is used to point to data of a type which is specified by another member. In the C coding a cast is used to obtain a pointer to the desired data type.

PDBLib supports this same practice by allowing the programmer to override the type of a member as given in the structure definition (see **PD_defstr**) by supplying the name of a member, whose type must be “char *”, which will contain an ASCII string specifying the actual type of the data to which the first member points.

The arguments to **PD_cast** are: *file*, a pointer to a **PDBfile**; *type*, an ASCII string containing the name of the data structure type in the PDB file; *memb*, an ASCII string containing the name of the member whose type is to be overridden; and *contr*, an ASCII string containing the name of the member (whose type must be “char *”) which will provide the actual type for *memb*.

The return value is a TRUE if the cast is successful; otherwise, FALSE is returned and the ASCII string **PD_err** contains any error message that was generated.

```
int PD_cast(PDBfile *file,
            char *type,
            char *memb,
            char *contr)
```

Example

```
#include “pdb.h”

   PDBfile *file;
   struct sample
      {char *type;
       int *a;};
         .
         .
         .
    PD_defstr(file, “sample”, “char *type”, “int *a”, LAST);
    PD_cast(file, “sample”, “a”, “type”);
         .
         .
         .
```

4.6 PD_CD

```
int PD_cd(PDBfile *file,
          char *dirname)
```

Change the current directory in the specified PDB file.

PDBLib supports an optional hierarchical directory structure inside PDB files. A directory or a variable in a directory may be specified by either a relative path or an absolute path. Slashes separate nodes in a path name. Absolute paths begin with a slash. Nodes consisting of two periods, "..", refer to the next higher level directory.

The arguments to **PD_cd** are: *file*, a pointer to a **PDBfile**; and *dirname*, an ASCII string containing the path name of the directory to change to.

If *dirname* is NULL or an empty string or a slash, it refers to the top level or root directory.

The return value is a TRUE if successful; otherwise, FALSE is returned and the ASCII string **PD_err** contains any error message that was generated.

See also **PD_ln**, **PD_ls**, **PD_mkdir**, and **PD_pwd**.

```
int PD_cd(PDBfile *file,
          char *dirname)
```

Example

```
#include “pdb.h”

   PDBfile *file;
         .
         .
         .
    if (PD_cd(file, “/animals/mammals”) == FALSE)
       printf(“%s”, PD_err);
         .
         .
         .
    if (PD_cd(file, “../reptiles”) == FALSE)
       printf(“%s”, PD_err);
         .
         .
         .
```

4.7 PD_CLOSE

```
int PD_close(PDBfile *file)
```

Close a PDB file. After all data is written to the PDB file, the structure chart and symbol table must be written out to the file and their disk addresses recorded in the file header. Without these operations the file cannot be read back in by PDBLib and all data is lost. All open PDB files must be **PD_close**'d before exiting the program.

A pointer, *file*, to the **PDBfile** associated with the PDB file must be passed to **PD_close**.

The function returns TRUE if the PDB file is correctly written and closed; otherwise, FALSE is returned and the ASCII string **PD_err** contains any error message that was generated.

```
int PD_close(PDBfile *file)
```

Example

```
#include “pdb.h”

   PDBfile *file;
            .
            .
            .
    if (PD_close(file) == FALSE)
       printf(“%s”, PD_err);
            .
            .
            .
```

4.8 PD_DEF_ATTRIBUTE

```
int PD_def_attribute(PDBfile *file,
                     char *attr,
                     char *type)
```

Define an attribute to the given PDB file. The model of an attribute in PDBLib is an entity that has a name and type. The two supported operations on attributes are to create them and to remove them. An entity in a PDB file can be assigned an attribute value simply by making a call which specifies the entity name, the attribute name, and the attribute value (which is determined by the type). The only association between an entry in a PDB file and any attribute is made by the name in the attribute table and the entry in the symbol table. In particular, this mechanism allows the application developer to define and use entities in a PDB file solely in terms of attributes.

The arguments to this function are: *file*, a pointer to a **PDBfile**; *name*, an ASCII string containing the name of the attribute being defined; and *type*, an ASCII string containing the name of the data type in the PDB file.

The return value is TRUE if successful, and FALSE otherwise.

See also **PD_rem_attribute**, **PD_set_attribute**, and **PD_get_attribute**.

```
int PD_def_attribute(PDBfile *file,
                     char *attr,
                     char *type)
```

Example

```
#include “pdb.h”

   PDBfile *file;
           .
           .
           .
    if (PD_def_attribute(file, “date”, “char *”) == FALSE)
       printf(“%s”, PD_err);
           .
           .
           .
```

4.9 PD_DEFENT

```
syment *PD_defent(PDBfile *file,
                  char *name,
                  char *outtype)
```

Define an entry in the symbol table of the PDB file specified by *file*. This function reserves space on disk but writes no data. The data can be written with later calls to **PD_write**, **PD_write_alt**, **PD_write_as**, or **PD_write_as_alt**.

The rationale for this function is to block out space in a PDB file corresponding to some logical layout of a piece of data. The data may not exist at the time the space is reserved or for some reason it may be desirable to write out the data in pieces. In any case if the type and shape of a variable is known at some point, an entry may be made in the PDB file without writing any data. The space may filled with other PDBLib calls at some later time.

The shape or dimensional information of the entry is a part of the name string. In this respect **PD_defent** behaves just like **PD_write**.

Input to this function is: *file*, a pointer to a **PDBfile**, *name*, an ASCII string containing the name of the variable and any dimensional information, and *outtype*, an ASCII string specifying the type of data in the file.

This function returns a symbol table entry (**syment**) pointer, if successful; otherwise, NULL is returned and the ASCII string **PD_err** contains any error message that was generated.

See also **PD_defent_alt**, **PD_write**, **PD_write_alt**, **PD_write_as**, **PD_write_as_alt**.

```
syment *PD_defent(PDBfile *file,
                  char *name,
                  char *outtype)
```

Example

```
#include "pdb.h"

    PDBfile *file;
    syment *ep;

          .
          .
          .
    if ((ep = PD_defent(file, "x(20)", "float")) == NULL)
       printf("%s", PD_err);
          .
          .
          .
```

Compare this with the example of **PD_write**.

4.10 PD_DEFENT_ALT

```
syment *PD_defent_alt(PDBfile *file,
                      char *name,
                      char *outtype,
                      int nd,
                      long *ind)
```

Define an entry in the symbol table of the PDB file specified by *file*. This function reserves space on disk but writes no data. The data can be written with later calls to **PD_write**, **PD_write_alt**, **PD_write_as**, or **PD_write_as_alt**.

This is an alternate form of **PD_defent**. The difference is that the dimension information is supplied via the *nd* and *ind* arguments instead of being a part of the name string. In this respect it behaves as **PD_write_alt** does.

The rationale for this function is to block out space in a PDB file corresponding to some logical layout of a piece of data. The data may not exist at the time the space is reserved or for some reason it may be desirable to write out the data in pieces. In any case if the type and shape of a variable is known at some point, an entry may be made in the PDB file without writing any data. The space may filled with other PDBLib calls at some later time.

Input to this function is: *file*, a pointer to a **PDBfile**, *name*, an ASCII string containing the name of the variable only, *outtype*, an ASCII string specifying the type of data in the file; *nd*, an integer specifying the number of dimensions; and *ind*, an array of long integers containing the minimum and maximum values of the index for each dimension pairwise.

This function returns a symbol table entry (**syment**) pointer, if successful; otherwise, NULL is returned and the ASCII string **PD_err** contains any error message that was generated.

See also **PD_defent**, **PD_write**, **PD_write_alt**, **PD_write_as**, **PD_write_as_alt**.

```
syment *PD_defent_alt(PDBfile *file,
                      char *name,
                      char *outtype,
                      int nd,
                      long *ind)
```

Example

```
#include “pdb.h”

    PDBfile *file;
    syment *ep;
    long ind[4];
          .
          .
          .
    ind[0] = 0L;
    ind[1] = 19L;
    ind[2] = -2L;
    ind[3] = 2L;
    if ((ep = PD_defent_alt(file, “x”, “float”, 2, ind)) == NULL)
       printf(“%s”, PD_err);
          .
          .
          .
```

Compare this with the example of **PD_write_alt** and note the absence of a two dimensional array in this call.

4.11 PD_DEFIX

```
defstr *PD_defix(PDBfile *file,
                 char *name,
                 long bytespitem,
                 int align,
                 int flg)
```

Define a primitive integral type (fixed point type) in the PDB file specified by *file*.

Input to **PD_defix** is: *file*, a pointer to a **PDBfile**; *name*, an ASCII string containing the name of the new data type; *bytespitem*, the number of bytes required for 1 item of the new type; *align*, the byte alignment for the type; and *flg*, a flag indicating whether the byte ordering of the type is normal or reverse ordered.

PDBLib supplies two #define'd constants which define the two ordering schemes used for fixed point types: **NORMAL_ORDER** and **REVERSE_ORDER**. **NORMAL_ORDER** means that the byte ordering from lowest to highest address as occurs on most CPU's. **REVERSE_ORDER** means that the byte order goes from highest to lowest address as happens with INTEL and other CPU's.

Compare this information with that found in the discussion of data conversion later in this manual.

A pointer to the new type's **defstr** is returned if the call is successful; otherwise, NULL is returned and the ASCII string **PD_err** contains any error message that was generated.

```
defstr *PD_defix(PDBfile *file,
                 char *name,
                 long bytespitem,
                 int align,
                 int flg)
```

Example

```
#include "pdb.h"

   PDBfile *file;
   defstr *ptr;
          .
          .
          .
    ptr = PD_defix(file, "int40", 5, 1, NORMAL_ORDER);
          .
          .
          .
```

4.12 PD_DEFLOAT

```
defstr *PD_defloat(PDBfile *file,
                   char *name,
                   long bytespitem,
                   int align,
                   int *ordr,
                   long expb,
                   long mantb,
                   long sbs,
                   long sbe,
                   long sbm,
                   long hmb,
                   long bias)
```

Define a new floating point type to the PDB file specified by *file*.

Input to **PD_defloat** is: *file*, a pointer to a **PDBfile**; n*ame*, an ASCII string containing the name of the variable in the PDB file; b*ytespitem*, the number of bytes required for an item of the new type; *align*, the byte alignment for this type; *ordr*, an array of *bytespitem* integers specifying the byte order; *expb*, the number of exponent bits; m*antb*, the number of mantissa bits; *sbs*, the position of the sign bit; *sbe*, the starting bit of the exponent; *sbm*, the starting bit of the mantissa; *hmb*, the value of the high order mantissa bit; and *bias*, the bias of the exponent.

Compare this information with that found in the discussion of data conversion later in this manual.

A pointer to the new type's **defstr** is returned if the call is successful; otherwise, NULL is returned and the ASCII string **PD_err** contains any error message that was generated.

```
defstr *PD_defloat(PDBfile *file,
                   char *name,
                   long bytespitem,
                   int align,
                   int *ordr,
                   long expb,
                   long mantb,
                   long sbs,
                   long sbe,
                   long sbm,
                   long hmb,
                   long bias)
```

Example

```
#include “pdb.h”

int ord_int24[] = {1, 3, 2};

   PDBfile *file;
   defstr *ptr;
          .
          .
          .
    ptr = PD_defloat(file, “fp24”, 3, 1, ord_int24,
                     7L, 16L, 0L, 1L, 8L, 0L, 0x3F)
          .
          .
          .
```

4.13 PD_DEFNCV

```
defstr *PD_defncv(PDBfile *file,
                  char *name,
                  long bytespitem,
                  int align)
```

Define a primitive type that will not undergo format conversion from platform to platform in the PDB file specified by *file*. Certain data types commonly defined in C programs are used as flags or character holders. With such data types the actual bit pattern contains the meaningful information. This information would be lost under a data conversion operation. This function provides users with a means to define primitive types which will not be converted under any circumstances and therefore preserve the meaningful bit patterns which constitute the intended data.

Input to **PD_defncv** is: *file*, a pointer to a **PDBfile**; *name*, an ASCII string containing the name of the new data type; *bytespitem*, the number of bytes required for 1 item of the new type; and *align*, the byte alignment for the type.

Compare this information with that found in the discussion of data conversion later in this manual.

A pointer to the new type's **defstr** is returned if the call is successful; otherwise, NULL is returned and the ASCII string **PD_err** contains any error message that was generated.

```
defstr *PD_defncv(PDBfile *file,
                  char *name,
                  long bytespitem,
                  int align)
```

Example

```
#include “pdb.h”

   PDBfile *file;
   defstr *ptr;
          .
          .
          .
/* define a type like the FORTRAN character*8 */
    ptr = PD_defncv(file, “char_8”, 8, 1);
          .
          .
          .
```

4.14 PD_DEFSTR

```
defstr *PD_defstr(PDBfile *file,
                  char *name,
                  char *mem1, ..., memn,
                  int *LAST)
```

Define a data structure for a PDB file. As a matter of programming efficiency and clarity it is useful to be able to refer to more complex structural units than the primitive types: short integers, integers, long integers, floating point numbers, double precision floating point numbers, and characters. Arrays do this in a very simple-minded way. Many modern languages support extended types or structures which allow the programmer to group diverse types of data together in a very sophisticated way.

PDBLib supports an extended data typing mechanism called a structure. A structure is a set of declarations of members. Each member is in turn a data type known to the system. Much of the style and usage of structures comes from the C struct. Because of the memory management features upon which PDBLib now depends, even members whose types are pointers are allowed. The only restrictions on member types are that they not be function pointers and that they be expressible without parentheses. Again any member which is a pointer must have its memory allocated by a SCORE memory management function or macro. See the Memory Management section near the beginning of this manual.

PD_defstr defines structures to the PDB system so that they can be read and written as a whole in a single statement. The members of the structure are processed and an entry in the structure chart is made. Then subsequent references to the new structure type are processed using information from the structure chart. The syntax by which members of a structure are specified is like that for C structs. The formal definition is given below ([ ] enclose optional elements). Self-referential structures are allowed providing the reference is through pointers (like C). The actual type name is used in the reference since PDBLib checks that all member types are already known or are the type being defined.

<member>	:=	<type> [*...*]<member name>[(<dimensions>)]
<type>	:=	<primitive type> | <derived type>
<member name>	:=	an ascii string representing the name of the member
<primitive type>	:=	short | integer | long | float | double | char
<derived type>	:=	any **PD_defstr**'d type
<dimensions>	:=	<integer> |
		<integer : integer> |
		<integer>, <dimensions> |
		<integer : integer> <dimensions>

Dimensions can be given in two ways. If a single number is specified for a given dimension, it is interpreted as the length of that dimension and the default offset value for the PDB file is taken as the minimum value. In this case the maximum value is the default offset plus the minimum value minus one. Alternatively, the minimum value followed by a colon and the maximum value may be specified. For example,

integer a(30,1:10)

The arguments to **PD_defstr** are: *file*, a pointer to a **PDBfile**; *name*, an ASCII string containing the name of the data structure type in the PDB file; and mem_i, a list of ASCII strings each representing the declaration of a member of a structure are defined above. **LAST** must terminate the list of members.

The return value is a pointer to the entry made in the structure chart if the call is successful; otherwise, NULL is returned and the ASCII string **PD_err** contains any error message that was generated.

```
defstr *PD_defstr(PDBfile *file,
                  char *name,
                  char *mem1, ..., memn,
                  int *LAST)
```

Example

```
#include "pdb.h"

   PDBfile *file;
   defstr *ptr;
   struct sample
      {float x[20];
       float y[20];
       int number;};
          .
          .
          .
    ptr = PD_defstr(file, "sample", "float x(20)", "float y(20)",
                          "int number", LAST);
          .
          .
          .
```

4.15 PD_DEFSTR_ALT

```
defstr *PD_defstr_alt(PDBfile *file,
                      char *name,
                      int nmemb,
                      char **members)
```

Define a data structure for a PDB file. As a matter of programming efficiency and clarity it is useful to be able to refer to more complex structural units than the primitive types: short integers, integers, long integers, floating point numbers, double precision floating point numbers, and characters. Arrays do this in a very simple-minded way. Many modern languages support extended types or structures which allow the programmer to group diverse types of data together in a very sophisticated way.

PDBLib supports an extended data typing mechanism called a structure. A structure is a set of declarations of members. Each member is in turn a data type known to the system. Much of the style and usage of structures comes from the C struct. Because of the memory management features upon which PDBLib now depends, even members whose types are pointers are allowed. The only restrictions on member types are that they not be function pointers and that they be expressible without parentheses. Again any member which is a pointer must have its memory allocated by a SCORE memory management function or macro. See the Memory Management section near the beginning of this manual.

PD_defstr defines structures to the PDB system so that they can be read and written as a whole in a single statement. The members of the structure are processed and an entry in the structure chart is made. Then subsequent references to the new structure type are processed using information from the structure chart. The syntax by which members of a structure are specified is like that for C structs. The formal definition is given below ([ ] enclose optional elements). Self-referential structures are allowed providing the reference is through pointers (like C). The actual type name is used in the reference since PDBLib checks that all member types are already known or are the type being defined.

<member>	:=	<type> [*...*]<member name>[(<dimensions>)]
<type>	:=	<primitive type> | <derived type>
<member name>	:=	an ascii string representing the name of the member
<primitive type>	:=	short | integer | long | float | double | char
<derived type>	:=	any **PD_defstr**'d type
<dimensions>	:=	<integer> |
		<integer : integer> |
		<integer>, <dimensions> |
		<integer : integer> <dimensions>

Dimensions can be given in two ways. If a single number is specified for a given dimension, it is interpreted as the length of that dimension and the default offset value for the PDB file is taken as the minimum value. In this case the maximum value is the default offset plus the minimum value minus one. Alternatively, the minimum value followed by a colon and the maximum value may be specified. For example,

```
integer a(30,1:10)
```

The arguments to **PD_defstr** are: *file*, a pointer to a **PDBfile**; *name*, an ASCII string containing the name of the data structure type in the PDB file; *nmemb*, an integer number of strings in the members array; and *members*, an array of ASCII strings each representing the declaration of a member of a structure are defined above.

The return value is a pointer to the entry made in the structure chart if the call is successful; otherwise, NULL is returned and the ASCII string **PD_err** contains any error message that was generated.

```
defstr *PD_defstr_alt(PDBfile *file,
                      char *name,
                      int nmemb,
                      char **members)
```

Example

```
#include “pdb.h”

   PDBfile *file;
   defstr *ptr;
   char **members;
   struct sample
      {float x[20];
       float y[20];
       int number;};
          .
          .
          .
    members = MAKE_N(char *, 3);
    members[0] = SC_strsave(“float x[20]”);
    members[1] = SC_strsave(“float y[20]”);
    members[2] = SC_strsave(“integer number”);
    ptr = PD_defstr_alt(file, “sample”, 3, members);

    SFREE(members[0]);
    SFREE(members[1]);
    SFREE(members[3]);
    SFREE(members);
          .
          .
          .
```

4.16 PD_FAMILY

```
PDBfile *PD_family(PDBfile *file, int flag)
```

This function checks to see whether the specified file has exceeded it size limit. If it has a new file is opened and returns. If not the given file pointer is returned. The flag is set to TRUE if you want **PD_family** to close the file it is given. Otherwise the application is responsible for closing the file

The arguments to this function are: *file*, the pointer to the PDBfile structure returned by a previous call to **PD_open**; and *flag* an integer value (either TRUE or FALSE).

This function returns a pointer to a PDBfile.

```
PDBfile *PD_family(PDBfile *file, int flag)
```

Example

```
#include "pdb.h"

   PDBfile *old, *new;
           .
           .
           .
    new = PD_family(old, FALSE);
    if (new != old)
       {PD_close(old);
             .
             .
             .
        old = new;};
           .
           .
           .
```

4.17 PD_FLUSH

```
int PD_flush(PDBfile *file)
```

This function writes out the information which describes the contents of the PDB file specified. Normally, **PD_close** calls this routine, but applications that want to protect themselves from system failures or other problems may chose to periodically use this function. After a successful return and until or unless more data is written to the file or space reserved for future writes, the PDB file is valid in the sense that if the application terminates unexpectedly before calling **PD_close**, the file can be **PD_open**'d successfully.

NOTE: this call does NOT obviate **PD_close**!

The argument to this function is *file*, the pointer to the PDBfile structure returned by a previous call to **PD_open**.

This function returns TRUE if successful and FALSE otherwise.

```
int PD_flush(PDBfile *file)
```

Example

```
#include “pdb.h”

   PDBfile *file;
            .
            .
            .
    if (PD_flush(file) == FALSE)
       printf(“%s”, PD_err);
            .
            .
            .
```

4.18 PD_GET_ATTRIBUTE

```
void *PD_get_attribute(PDBfile *file,
                       char *vr,
                       char *attr)
```

Return the value of the specified attribute for the named entity.

The model of an attribute in PDBLib is an entity that has a name and type. The two supported operations on attributes are to create them and to remove them. An entity in a PDB file can be assigned an attribute value simply by calling **PD_set_attribute**. The only association between an entry in a PDB file and any attribute is made by the name in the attribute table and the entry in the symbol table. In particular, this mechanism allows the application developer to define and use entities in a PDB file solely in terms of attributes.

Attribute values are always assigned by reference and **PD_get_attribute** returns them the same way. The application may have to make a cast on the returned pointer.

The arguments to this function are: *file*, a pointer to a **PDBfile**; *vr*, an ASCII string containing the name of an entity in the PDB file; and *attr*, an ASCII string containing the name of the attribute being sought.

The return value is a pointer to the value of the attribute if one exists and NULL otherwise.

See also **PD_def_attribute**, **PD_rem_attribute**, and **PD_set_attribute**.

```
void *PD_get_attribute(PDBfile *file,
                       char *vr,
                       char *attr)
```

Example

```
#include “pdb.h”

   PDBfile *file;
   char *dt;
           .
           .
           .
    dt = (char *) PD_get_attribute(file, “foo”, “date”);
       printf(“%s”, PD_err);
           .
           .
           .
```

4.19 PD_LN

```
int PD_ln(PDBfile *file,
          char *var,
          char *link)
```

Create a link to a variable in the specified PDB file.

PDBLib supports an optional hierarchical directory structure inside PDB files. A directory or a variable in a directory may be specified by either a relative path or an absolute path. Slashes separate nodes in a path name. Absolute paths begin with a slash. Nodes consisting of two periods, "..", refer to the next higher level directory.

The arguments to **PD_ln** are: *file*, a pointer to a **PDBfile**; *var*, an ASCII string containing the path name of an existing variable; and *link*, an ASCII string containing the path name of the new link.

The return value is a TRUE if successful; otherwise, FALSE is returned and the ASCII string **PD_err** contains any error message that was generated.

See also **PD_cd**, **PD_ls**, **PD_mkdir**, and **PD_pwd**.

```
int PD_ln(PDBfile *file,
          char *var,
          char *link)
```

Example

```
#include “pdb.h”

   PDBfile *file;
         .
         .
         .
    if (PD_ln(file, “/animals/mammals/chimpanzee”, “/chimp”) == FALSE)
       printf(“%s”, PD_err);
         .
         .
         .
```

4.20 PD_LS

```
char **PD_ls(PDBfile *file,
             char *path,
             char *type,
             int *num)
```

Return a list of names of entries (variables and directories) in PDB file *file* that are of type *type* and that are in the directory and match the variable name pattern specified by *path*.

PDBLib supports an optional hierarchical directory structure inside PDB files. A directory or a variable in a directory may be specified by either a relative path or an absolute path. Slashes separate nodes in a path name. Absolute paths begin with a slash. Nodes consisting of two periods, "..", refer to the next higher level directory.

The arguments to **PD_ls** are: *file*, a pointer to a **PDBfile**; *path*, an ASCII string containing the path name of the directory to search and/or the variable name pattern to match; *type*, an ASCII string containing the type of entries to return; and *num*, a pointer to an integer to contain the number of entries returned.

If *path* is NULL, the contents of the current directory are listed. If *type* is NULL, all types are returned.

The terminal node of *path* may contain meta characters "*" and "?". Each "*" matches any zero or more characters and each "?" matches any single character.

For the sake of efficiency, the returned names are not duplicated. That is, the caller should not free the space associated with each of the individual strings, but should free the char ** pointer. This should be done using the **SFREE** macro as shown in the example.

The return value is a pointer to an array of strings, if successful; otherwise, NULL is returned and the ASCII string **PD_err** contains any error message that was generated.

See also **PD_cd**, **PD_ln**, **PD_mkdir**, and **PD_pwd**.

```
int PD_ls(PDBfile *file,
          char *path,
          char *type,
          int *num)
```

Example

```
#include “pdb.h”

   PDBfile *file;
   char **list;
   int num;
         .
         .
         .
/* get a list of all directories in the current directory */
    list = PD_ls(file, NULL, “Directory”, &num);
    if (list == NULL)
       printf(“%s”, PD_err);
         .
         .
         .
    SFREE(list);
         .
         .
         .
/* get a list of the variables of type char * in directory animals */
    list = PD_ls(file, “animals”, “char *”, &num);
    if (list == NULL)
       printf(“%s”, PD_err);
         .
         .
         .
    SFREE(list);
         .
         .
         .
```

4.21 PD_MKDIR

```
int PD_mkdir(PDBfile *file,
             char *dirname)
```

Create a new directory in the specified PDB file.

PDBLib supports an optional hierarchical directory structure inside PDB files. A directory or a variable in a directory may be specified by either a relative path or an absolute path. Slashes separate nodes in a path name. Absolute paths begin with a slash. Nodes consisting of two periods, "..", refer to the next higher level directory.

The arguments to **PD_mkdir** are: *file*, a pointer to a **PDBfile** and *dirname*, an ASCII string containing the path name of the new directory.

The root directory, "/", does not have to be created.

The return value is a TRUE if successful; otherwise, FALSE is returned and the ASCII string **PD_err** contains any error message that was generated.

See also **PD_cd**, **PD_ln, PD_ls**, and **PD_pwd**.

```
int PD_mkdir(PDBfile *file,
             char *dirname)
```

Example

```
#include “pdb.h”

   PDBfile *file;
         .
         .
         .
    if (PD_mkdir(file, “/animals/mammals) == FALSE)
       printf(“%s”, PD_err);
         .
         .
         .
```

4.22 PD_OPEN

```
PDBfile *PD_open(char *filename,
                 char *mode)
```

Open an existing PDB file or create a new PDB file. Depending on the value of the *mode* argument, PDBLib attempts to open the file *filename* in read-only binary mode, open the file in append binary mode, or create a new file in read-write binary mode. Any string which begins with “r” causes the file to be opened in read-only mode, any string beginning with “a” causes the file to be opened in append mode, and any string beginning with “w” causes a file to be created in read-write mode. Next the beginning of the file is searched for the header which identifies the file as having been generated by PDBLib. The addresses of the structure chart and symbol table are then sought.

The structure chart from the file is read in. The structure chart contains information about data types (e.g. floats), their sizes in bytes and their structures if any. By default there are six primitive data types that PDBLib knows about: short integers, integers, long integers, floating point numbers, double precision floating point numbers, characters, and pointers. The sizes of these types varies from machine to machine, but PDBLib hides this from the user.

The symbol table from the file is read in. The symbol table contains the list of variables in the file, their types as defined in the structure chart, and dimensioning information for arrays. Each read from the file first consults the symbol table to see if the requested variable is present in the PDB file.

Both the structure chart and the symbol table are implemented as hash tables, although their shapes are different. This makes lookups as efficient as possible given an unknown amount of data in the file.

The arguments to **PD_open** are: *filename*, an ASCII string, which is the name of the file to be created or opened; and *mode*, an ASCII string, which is the mode (either “w” for create, “r” for read-only or “a” for append).

The function returns a pointer to a **PDBfile**. This **PDBfile** identifies the particular file to PDBLib. As such, if it is overwritten the file is lost. The number of PDB files which can be open simultaneously is machine or operating system dependent, but each open file has a unique **PDBfile** associated with it.

If any aspect of the PDB file opening process fails a NULL pointer is returned and the ASCII string **PD_err** contains any error message that was generated.

```
PDBfile *PD_open(char *filename,
                 char *mode)
```

Example

```
#include “pdb.h”

   PDBfile *file;
          .
          .
          .
   if ((file = PD_open(“filenam”, “r”)) == NULL)
      printf(“%s”, PD_err);
          .
          .
          .
```

4.23 PD_PWD

```
char *PD_pwd(PDBfile *file)
```

Return the current directory for the specified PDB file.

PDBLib supports an optional hierarchical directory structure inside PDB files. A directory or a variable in a directory may be specified by either a relative path or an absolute path. Slashes separate nodes in a path name. Absolute paths begin with a slash. Nodes consisting of two periods, "..", refer to the next higher level directory.

The argument to **PD_pwd** is *file*, a pointer to a **PDBfile**.

If no directory has been created, "/" is returned.

The return value is a pointer to a string containing the path name of the current directory if successful; otherwise, NULL is returned and the ASCII string **PD_err** contains any error message that was generated.

See also **PD_cd**, **PD_ln, PD_ls**, and **PD_mkdir**.

int PD_pwd(PDBfile *file)

Example

```
#include “pdb.h”

   PDBfile *file;
   char *dirname;
         .
         .
         .
    if ((dirname = PD_pwd(file)) == NULL)
       printf(“%s”, PD_err);
         .
         .
         .
```

4.24 PD_READ

```
int PD_read(PDBfile *file,
            char *name,
            void *var)
```

Read all or part of a data entry from an open PDB file. The symbol table of the given PDB file is searched for the given name and if it is found the information there is used to read the proper number of bytes from the file, do any conversions, and put the result in memory pointed to by *var*.

The arguments to **PD_read** are: *file*, a pointer to a **PDBfile** which designates the PDB file from which to attempt the read; *name*, an ASCII string containing the specification of data to be read; and *var*, a pointer to the location where the data is to be placed.

Note: In each **PD_read** operation, the type of *var* must be a pointer to the type of the variable *name*.

Note: PDBLib can only read part of an entry if the type of the terminal node is primitive or a structure which contains no indirections and whose descendant members contain no indirections.

Note: When reading part of a variable, especially a structured variable, the path to the desired part must contain one array reference for each level of indirection traversed.

The return value is TRUE, if successful; otherwise, FALSE is returned and the ASCII string **PD_err** contains any error messages.

See also **PD_read_alt**, **PD_read_as**, **PD_read_as_alt**.

```
int PD_read(PDBfile *file,
            char *name,
            void *var)
```

Example

```
#include “pdb.h”

    PDBfile *file;
    float x[20];
          .
          .
          .
    if (PD_read(file, “x”, x) == FALSE)
       printf(“%s”, PD_err);
          .
          .
          .
```

4.25 PD_READ_ALT

```
int PD_read_alt(PDBfile *file,
                char *name,
                void *var,
                long *ind)
```

Read all or part of a data entry from an open PDB file. The symbol table of the given PDB file is searched for the given name and if it is found the information there is used to read the proper number of bytes from the file, do any conversions, and put the result in memory pointed to by *var*.

The arguments to **PD_read_alt** are: *file*, a pointer to a **PDBfile** which designates the PDB file from which to attempt the read; *name*, an ASCII string containing the specification of the data to be read; *var*, a pointer to the location where the data is to be placed; and *ind*, an array of long integers consisting of three indexes (start, stop, and step) for each dimension of the entry.

Note: In each **PD_read_alt** operation, the type of *var* must be a pointer to the type of the variable *name*.

Note: PDBLib can only read part of an entry if the type of the terminal node is primitive or a structure which contains no indirections and whose descendant members contain no indirections.

Note: When reading part of a variable, especially a structured variable, the path to the desired part must contain one array reference for each level of indirection traversed.

The return value is TRUE, if successful; otherwise, FALSE is returned and the ASCII string **PD_err** contains any error messages.

See also **PD_read**, **PD_read_as**, **PD_read_as_alt**.

```
int PD_read_alt(PDBfile *file,
                char *name,
                void *var,
                long *ind)
```

Example

```
#include “pdb.h”

    PDBfile *file;
    long ind[3];
    float x[20];
          .
          .
          .
    ind[0] = 3;
    ind[1] = 18;
    ind[2] = 2;
    if (PD_read_alt(file, “x”, x, ind) == FALSE)
       printf(“%s”, PD_err);
          .
          .
          .
```

4.26 PD_READ_AS

```
int PD_read_as(PDBfile *file,
               char *name,
               char *type,
               void *var)
```

Read data from an open PDB file. The symbol table of the given PDB file is searched for the given name and if it is found the information there is used with the type specified by *type* to read the proper number of bytes from the file, do any conversions, and put the result in memory pointed to by *var*. The type specified overrides the type in the symbol table entry as far as deciding on data conversions goes.

This function is generally used to read floats as doubles and so on. However with sufficient care and understanding of both the file data and C data structuring, it can be used to transmute structured data.

The arguments to **PD_read_as** are: *file*, a pointer to a **PDBfile** which designates the PDB file from which to attempt the read; *name*, an ASCII string containing the specification for the data to be read; *type*, an ASCII string containing the type of the data desired; and *var*, a pointer to the location where the data is to be placed.

Note: In each **PD_read_as** operation, the type of *var* must be a pointer to the type specified by *type*.

Note: PDBLib can only read part of an entry if the type of the terminal node is primitive or a structure which contains no indirections and whose descendant members contain no indirections.

Note: When reading part of a variable, especially a structured variable, the path to the desired part must contain one array reference for each level of indirection traversed.

The return value is TRUE, if successful; otherwise, FALSE is returned and the ASCII string **PD_err** contains any error messages.

See also **PD_read**, **PD_read_alt**, **PD_read_as_alt**.

```
int PD_read_as(PDBfile *file,
               char *name,
               char *type,
               void *var)
```

Example

```
#include “pdb.h”

    PDBfile *file;
    float x[20];
          .
          .
          .
/* x is a double in the file */
    if (PD_read_as(file, “x”, “float”, x) == FALSE)
       printf(“%s”, PD_err);
          .
          .
          .
```

4.27 PD_READ_AS_ALT

```
int PD_read_as_alt(PDBfile *file,
                   char *name,
                   char *type,
                   void *var,
                   long *ind)
```

Read all or part of a data entry from an open PDB file. The symbol table of the given PDB file is searched for the given name and if it is found the information there is used with the type specified by *type* to read the proper number of bytes from the file, do any conversions, and put the result in memory pointed to by *var*. The type specified overrides the type in the symbol table entry as far as deciding on data conversions goes.

This function is generally used to read floats as doubles and so on. However with sufficient care and understanding of both the file data and C data structuring, it can be used to transmute structured data.

The arguments to **PD_read_as_alt** are: *file*, a pointer to a **PDBfile** which designates the PDB file from which to attempt the read; *name*, an ASCII string containing the specification of the data to be read; *type*, an ASCII string containing the type of the data desired; *var*, a pointer to the location where the data is to be placed; and *ind*, an array of long integers consisting of three indexes (start, stop, and step) for each dimension of the entry.

Note: In each **PD_read_as_alt** operation, the type of *var* must be a pointer to the type specified by *type*.

Note: PDBLib can only read part of an entry if the type of the terminal node is primitive or a structure which contains no indirections and whose descendant members contain no indirections.

Note: When reading part of a variable, especially a structured variable, the path to the desired part must contain one array reference for each level of indirection traversed.

The return value is TRUE, if successful; otherwise, FALSE is returned and the ASCII string **PD_err** contains any error messages.

See also **PD_read**, **PD_read_as**, **PD_read_alt**.

```
int PD_read_as_alt(PDBfile *file,
                   char *name,
                   char *type,
                   void *var,
                   long *ind)
```

Example

```
#include “pdb.h”

    PDBfile *file;
    long ind[3];
    float x[20];
          .
          .
          .
    ind[0] = 2;
    ind[1] = 10;
    ind[2] = 2;

/* x is a double in the file */
    if (PD_read_as_alt(file, “x”, “float”, x, ind) == FALSE)
       printf(“%s”, PD_err);
          .
          .
          .
```

4.28 PD_RESET_PTR_LIST

```
int PD_reset_ptr_list(PDBfile *file)
```

Free the list of pointers which the PDB file *file* knows about. This includes both pointers in memory acquired during write operations and pointers in the file acquired during read operations.

Rationale: When reading or writing indirectly referenced data, PDBLib, maintains an array of pointers encountered in write operations and an array of pointers encountered in read operations. This is done on a per file basis. These arrays are the basis on which PDBLIb can determine how to preserve the connectivity of data trees when they are moved between memory and files. Because of the difference between memory and disk files, it is important for applications to be able to clear out these arrays and start over. See the discussion on Using Pointers earlier in the manual.

The argument to this function is *file* a pointer to a **PDBfile**.

The return value is TRUE if successful, and FALSE otherwise.

```
int PD_reset_ptr_list(PDBfile *file)
```

Example

```
#include “pdb.h”

   PDBfile *file;
           .
           .
           .
    if (PD_reset_ptr_list(file) == FALSE)
       printf(“%s”, PD_err);
           .
           .
```

4.29 PD_REM_ATTRIBUTE

```
int PD_rem_attribute(PDBfile *file,
                     char *attr)
```

Remove the specified attribute. Any entities which have a value for this attribute will have it removed by PDBLib.

The model of an attribute in PDBLib is an entity that has a name and type. The two supported operations on attributes are to create them and to remove them. An entity in a PDB file can be assigned an attribute value simply by calling **PD_set_attribute**. The only association between an entry in a PDB file and any attribute is made by the name in the attribute table and the entry in the symbol table. In particular, this mechanism allows the application developer to define and use entities in a PDB file solely in terms of attributes.

The arguments to this function are: *file*, a pointer to a **PDBfile**; and *attr*, an ASCII string containing the name of the attribute being removed.

The return value is TRUE if successful, and FALSE otherwise.

See also **PD_def_attribute**, **PD_get_attribute**, and **PD_set_attribute**.

```
int PD_rem_attribute(PDBfile *file,
                     char *attr)
```

Example

```
#include "pdb.h"

   PDBfile *file;
           .
           .
           .
    if (PD_rem_attribute(file, "date") == FALSE)
       printf("%s", PD_err);
           .
           .
           .
```

4.30 PD_SET_ATTRIBUTE

```
int PD_set_attribute(PDBfile *file,
                     char *vr,
                     char *attr,
                     void *vl)
```

Set the value of the specified attribute for the named entity. Attribute values are always assigned by reference.

The model of an attribute in PDBLib is an entity that has a name and type. The two supported operations on attributes are to create them and to remove them. An entity in a PDB file can be assigned an attribute value simply by calling **PD_set_attribute**. The only association between an entry in a PDB file and any attribute is made by the name in the attribute table and the entry in the symbol table. In particular, this mechanism allows the application developer to define and use entities in a PDB file solely in terms of attributes.

The arguments to this function are: *file*, a pointer to a **PDBfile**; *vr*, an ASCII string containing the name of an entity in the PDB file; *attr*, an ASCII string containing the name of the attribute being set; and *vl*, a pointer to data whose type matches the attribute type.

The return value is TRUE if successful and FALSE otherwise.

See also **PD_def_attribute**, **PD_rem_attribute**, and **PD_get_attribute**.

```
int PD_set_attribute(PDBfile *file,
                     char *vr,
                     char *attr,
                     void *vl)
```

Example

```
#include “pdb.h”

   PDBfile *file;
   char *dt;
           .
           .
           .
    dt = SC_strsave(“Mon March 23, 1921”);
    if (PD_set_attribute(file, “foo”, “date”, dt) == FALSE)
       printf(“%s”, PD_err);
           .
           .
           .
```

4.31 PD_TARGET

```
int PD_target(data_standard *std,
              data_alignment *align)
```

Write the next PDB file according to the specified data standard and alignment. PDBLib has a general binary data conversion mechanism called parametrized data conversion (PDC). The an integer type is described by one set of parameters and a floating point type is described by another. A general purpose conversion routine takes the description of the input type and a description of the desired output type and does the conversion. In this way, PDBLib avoids an N^2 increase in data conversion routines as it ports to new machines. In fact, the number of data standards and alignments grows more slowly than N because many machines share common formats.

An additional advantage to PDC is that by specifying a format involving the minimal number of bits to represent the data for a file, PDBLib can carry out a large class of data compressions.

For programmer convenience, PDBLib carries several data standards and alignments. Data standards: **IEEEA_STD**, **IEEEB_STD**, **IEEEC_STD**, **INTELA_STD**, **INTELB_STD**, **VAX_STD**, and **CRAY_STD**. Data alignments: **DEF_ALIGNMENT**, **SPARC_ALIGNMENT**, **MIPS_ALIGNMENT**, **M68000_ALIGNMENT**, **INTEL_ALIGNMENT**, and **CRAY_ALIGNMENT**.

Some common configurations are:

Motorola	- PD_target(&IEEEA_STD, &M68000_ALIGNMENT)
SPARC	- PD_target(&IEEEA_STD, &SPARC_ALIGNMENT)
MIPS, SGI	- PD_target(&IEEEA_STD, &MIPS_ALIGNMENT)
IBM RS6000	- PD_target(&IEEEA_STD, &RS6000_ALIGNMENT)
Mac/Think C	- PD_target(&IEEEB_STD, &M68000_ALIGNMENT)
Mac/MPW	- PD_target(&IEEEC_STD, &M68000_ALIGNMENT)
DOS	- PD_target(&INTELA_STD, &INTEL_ALIGNMENT)
Intel 80x86 UNIX	- PD_target(&INTELB_STD, &INTEL_ALIGNMENT)
DEC Vax	- PD_target(&VAX_STD, &DEF_ALIGNMENT)
DEC 3100	- PD_target(&INTELB_STD, &MIPS_ALIGNMENT)
UNICOS Cray	- PD_target(&CRAY_STD, &UNICOS_ALIGNMENT)

The argument, **std**, is a pointer to a **data_standard** structure, and the argument, *align*, is a pointer to a **data_alignment** structure. See the section on Data Structures.

The return value is TRUE, if successful; otherwise, FALSE is returned and the ASCII string **PD_err** contains any error messages.

```
int PD_target(data_standard *std,
              data_alignment *align)
```

Example

```
#include "pdb.h"

void test_target(tgt, base, n, fname, datfile)
   char *tgt, *base;
   int n;
   char *fname, *datfile;
   {if (tgt != NULL)

/* for DOS machines */
       {if (strcmp(tgt, "dos") == 0)
           PD_target(&INTEL_STD, &INTEL_ALIGNMENT);

/* for CRAY computers */
        else if (strcmp(tgt, "cray") == 0)
           PD_target(&CRAY_STD, &UNICOS_ALIGNMENT);

/* for DEC VAX machines */
        else if (strcmp(tgt, "vax") == 0)
           PD_target(&VAX_STD, &DEF_ALIGNMENT);

/* for MIPS based machines */
        else if (strcmp(tgt, "mips") == 0)
           PD_target(&IEEEA_STD, &MIPS_ALIGNMENT);

/* for standard M68000 machines */
        else if (strcmp(tgt, "sun3") == 0)
           PD_target(&IEEEA_STD, &M68000_ALIGNMENT);

/* for SPARC machines */
        else if (strcmp(tgt, "sun4") == 0)
           PD_target(&IEEEA_STD, &SPARC_ALIGNMENT);

/* for Macintosh */
        else if (strcmp(tgt, "mac") == 0)
```

```
        PD_target(&IEEEB_STD, &M68000_ALIGNMENT);

     sprintf(fname, “%s-%s.rs%d”, base, tgt, n);
     sprintf(datfile, “%s-%s.db%d”, base, tgt, n);}
else
    {sprintf(fname, “%s-nat.rs%d”, base, n);
     sprintf(datfile, “%s-nat.db%d”, base, n);};

return;}
```

4.32 PD_TYPEDEF

```
defstr *PD_typedef(PDBfile *file,
                   char *oname,
                   char *tname)
```

Define a an alternate name for an existing type. The intended use of this function is to allow users to make their PDB data types match their C types as closely as possible. It does this by mimicking the C typedef mechanism in a limited way. More accurately it provides an aliasing capability. This can be used in conjunction with either **PD_defix** or **PD_defloat** to install a definition of a data type in the host chart (**PD_defix** and **PD_defloat** define their types to the file chart only).

Input to **PD_defix** is: *file*, a pointer to a **PDBfile**; *oname*, an ASCII string containing the name of the original data type; *tname*, an ASCII string containing the name of the alias.

A pointer to the original type's **defstr** is returned if the call is successful; otherwise, NULL is returned and the ASCII string **PD_err** contains any error message that was generated.

```
defstr *PD_typedef(PDBfile *file,
                   char *oname,
                   char *tname)
```

Example

```
#include “pdb.h”

   PDBfile *file;
   defstr *ptr;
          .
          .
          .
/* define “enum” as an alias for “integer” */
    ptr = PD_typedef(file, “integer”, “enum”);
          .
          .
```

4.33 PD_WRITE

```
int PD_write(PDBfile *file,
             char *name,
             char *type,
             void *var)
```

Write data to a PDB file. If an entry already exists in the file, the data overwrites the specified file data; otherwise, before writing data to the PDB file an entry is prepared for the symbol table consisting of the name, the type, the dimension information, the disk address to which the data will be written, and the total number of bytes as computed with the help of the structure chart. After the entry is installed in the symbol table the data from memory is converted (only if the target machine type is different from the current machine type) and then written out to disk starting at the current disk address.

The primitive data types which the PDBLib system knows about by default are: “short”, “integer”, “long”, “float”, “double”, and “char” for short integer, integer, long integer, floating point or real number, double precision floating point number, and character or single byte respectively. Additional types may be added using **PD_defstr**.

PDBLib supports arbitrary levels of indirections. This means that all types of pointers (except function pointers) can be traced down to the actual data to which they point and that data will be written out into the PDB file in such a way that the read operations can reconstruct the data as it exists prior to the write operation. There is one crucial restriction. That is that the memory associated with any pointer must have been allocated using a SCORE memory management function or macro. See the memory management section near the beginning of this document.

Rationale: When writing out scalar variables (i.e. non-dimensioned variables - structured variables are scalars unless specifically dimensioned) this function is the most convenient to use since it involves no variable argument list and hence no worries about terminating the list. Another situation, which is more common than expected, in which **PD_write** would be preferred is when it is desirable to make entries in a PDB file which do not correspond to any variables in the application program. Since string manipulations might be involved in preparing the name under which to write the data, coding in the dimensional information is not any less efficient.

Dimensions can be given in two ways. If the variable was not previously defined and a single number is specified for a given dimension, it is interpreted as the length of that dimension and the default offset value for the PDB file is taken as the minimum value. In this case the maximum value is the default offset plus the minimum value minus one. (If the variable was previously defined, a single number is interpreted as a reference to that one dimension index value.) Alternatively, the minimum value, maximum value, and stride (separated by colons) may be specified. The stride is optional and defaults to 1.

The arguments to **PD_write** are: *file*, a pointer to a **PDBfile** which designates the PDB file to which to attempt to write; *name*, an ASCII string containing the name of the variable to install in the symbol table; *type*, an ASCII string specifying the variable type; and *var*, a pointer to the data to be written. This pointer must be consistent with the type specified, that is it must be a pointer to data with type, *type*. For example:

```
char **s, **t;
integer *u;

PD_write(file, "s", "char **", &s);
PD_write(file, "t(3)", "char *", t);
PD_write(file, "u(30,1:10)", "integer", u);
```

The dimension information is encoded in the ASCII string, *name*, as if in a FORTRAN dimension statement.

Note: PDBLib can only write part of an entry if the type of the terminal node is primitive or a structure which contains no indirections and whose descendant members contain no indirections.

Note: When writing part of a variable, especially a structured variable, the path to the desired part must contain one array reference for each level of indirection traversed.

The return value is TRUE if successful; otherwise, FALSE is returned and the ASCII string **PD_err** contains any error messages.

See also **PD_write_alt** and **PD_defent**.

```
int PD_write(PDBfile *file,
             char *name,
             char *type,
             void *var)
```

Example

```
#include “pdb.h”

    PDBfile *file;
    float x[20];
          .
          .
          .
    if (!PD_write(file, “x(20)”, “float”, x))
       printf(“%s”, PD_err);
          .
          .
          .
```

4.34 PD_WRITE_ALT

```
int PD_write_alt(PDBfile *file,
                 char *name,
                 char *type,
                 void *var,
                 int nd,
                 long *ind)
```

Write data to a PDB file. This is an alternate form to **PD_write**. If an entry already exists in the file, the data overwrites the specified file data; otherwise, before writing data to the PDB file an entry is prepared for the symbol table consisting of the name, the type, the dimension information, the disk address to which the data will be written, and the total number of bytes as computed with the help of the structure chart. After the entry is installed in the symbol table the data from memory is converted (only if the target machine type is different from the current machine type) and then written out to disk starting at the current disk address.

The primitive data types which the PDBLib system knows about by default are: “short”, “integer”, “long”, “float”, “double”, and “char” for short integer, integer, long integer, floating point or real number, double precision floating point number, and character or single byte respectively. Additional types may be added using **PD_defstr**.

PDBLib supports arbitrary levels of indirections. This means that, subject to the restrictions spelled out in the section on rules, pointers (except function pointers) can be traced down to the actual data to which they point and that data will be written out into the PDB file in such a way that the read operations can reconstruct the data as it exists prior to the write operation. There is one crucial restriction. That is that the memory associated with any pointer must have been allocated by a SCORE memory management function or macro. See the Memory Management section near the beginning of this document.

The rationale for this function is that in some situations, it is desirable to be able to specify the dimensions without building them into an ASCII string.

The arguments to **PD_write_alt** are: *file*, a pointer to a **PDBfile** which designates the PDB file to which to attempt to write; *name*, an ASCII string containing the name of the variable to install in the symbol table; *type*, an ASCII string specifying the variable type; *var*, a pointer to the data to be written; *nd*, then number of dimensions for the variable; and *ind*, an array of long integers containing (min, max, stride) triples specifying the ranges and strides of the dimensions. The pointer, *var*, must be consistent with the type specified, that is it must be a pointer to data with type, *type*. For example:

```
char **s, **t;
integer *u;
PD_write_alt(file, “s”, “char **”, &s, ...);
PD_write_alt(file, “t”, “char *”, t, ...);
PD_write_alt(file, “u”, “integer”, u, ...);
```

Note: PDBLib can only write part of an entry if the type of the terminal node is primitive or a structure which contains no indirections and whose descendant members contain no indirections.

Note: When writing part of a variable, especially a structured variable, the path to the desired part must contain one array reference for each level of indirection traversed.

The return value is TRUE if successful; otherwise, FALSE is returned and the ASCII string **PD_err** contains any error messages.

```
int PD_write_alt(PDBfile *file,
                 char *name,
                 char *type,
                 void *var,
                 int nd,
                 long *ind)
```

Example

```
#include “pdb.h”

    PDBfile *file;
    long ind[6];
    float x[20][5];
           .
           .
           .
    ind[0] = 0L;
    ind[1] = 19L;
    ind[2] = 1L;
    ind[3] = -2L;
    ind[4] = 2L;
    ind[5] = 1L;
    if (!PD_write_alt(file, “x”, “float”, x, 2, ind))
       printf(“%s”, PD_err);
           .
           .
           .
```

4.35 PD_WRITE_AS

```
int PD_write_as(PDBfile *file,
                char *name,
                char *intype,
                char *outtype,
                void *var)
```

Write the data pointed to by *var* under *name* and with *intype* in PDB file *file* as data of *outtype*.

The rationale for this function is that in some situations, it is desirable to not only convert the formats of data of a specified type, but to convert between types. An example that occurs in practice often enough is converting a 32 bit int to a 32 bit long on a machine which only has a 16 bit int.

Input to **PD_write_as** is: *file*, a pointer to a **PDBfile**; *name*, an ASCII string containing the name of the variable in the PDB file; *intype*, an ASCII string containing the type of the variable; *outtype*, an ASCII string containing the output type of the variable; and *var*, a pointer to the location where the data is to be stored in memory.

Note: PDBLib can only write part of an entry if the type of the terminal node is primitive or a structure which contains no indirections and whose descendant members contain no indirections.

Note: When writing part of a variable, especially a structured variable, the path to the desired part must contain one array reference for each level of indirection traversed.

PD_write_as returns TRUE if successful; otherwise, FALSE is returned and the ASCII string **PD_err** contains any error messages.

See also **PD_write_as_alt** and **PD_defent_alt**.

```
int PD_write_as(PDBfile *file,
                char *name,
                char *intype,
                char *outtype,
                void *var)
```

Example

```
#include “pdb.h”

    PDBfile *file;
    float x[20];
          .
          .
          .
    if (!PD_write_as(file, “x(20)”, “float”, “double”, x))
       printf(“%s”, PD_err);
          .
          .
          .
```

4.36 PD_WRITE_AS_ALT

```
int PD_write_as_alt(PDBfile *file,
                    char *name,
                    char *intype,
                    char *outtype,
                    void *var,
                    int nd,
                    long *ind)
```

Write the data pointed to by *var* under *name* and with *intype* in PDB file *file* as data of *out-type*. This is an alternate form of **PD_write_as**.

The rationale for this function is that in some situations, it is desirable to be able to specify the dimensions without building them into an ASCII string.

Input to **PD_write_as_alt** is: *file*, a pointer to a **PDBfile**; *name*, an ASCII string containing the name of the variable in the PDB file; *intype*, an ASCII string containing the type of the variable; *outtype*, an ASCII string containing the output type of the variable; *var*, a pointer to the location where the data is to be stored in memory; *nd*, an integer number of dimensions; and *ind*, an array of long integers specifying the ranges of each dimension (min, max, stride).

Note: PDBLib can only write part of an entry if the type of the terminal node is primitive or a structure which contains no indirections and whose descendant members contain no indirections.

Note: When writing part of a variable, especially a structured variable, the path to the desired part must contain one array reference for each level of indirection traversed.

PD_write_as_alt returns TRUE if successful; otherwise, FALSE is returned and the ASCII string **PD_err** contains any error messages.

```
int PD_write_as_alt(PDBfile *file,
                    char *name,
                    char *intype,
                    char *outtype,
                    void *var,
                    int nd,
                    long *ind)
```

Example

```
#include “pdb.h”

    PDBfile *file;
    long ind[6];
    float x[20][5];
          .
          .
          .
    ind[0] = 0L;
    ind[1] = 19L;
    ind[2] = 1L;
    ind[3] = -2L;
    ind[4] = 2L;
    ind[5] = 1L;
    if (!PD_write_as_alt(file, “x”, “float”, “double”, x, 2, ind))
       printf(“%s”, PD_err);
          .
          .
          .
```

5.0 Summary of the FORTRAN API

There is a hierarchy of routines in PDBLib from high to low level. The high level routines form the API while the lower level routines are modularized to perform the actual work. It should be noted that the lower level routines are sufficiently well modularized so as to make it possible to build entirely different API's for PDBLib.

The high level PDBLib routines have a strict naming convention. All routines in the FORTRAN API begin with 'PF'.

Note: many of these functions return integer values. The implicit typing convention in FORTRAN would indicate that they return real values. Application programs must explicitly type these functions as integers.

When an error condition is detected by PDBLib it saves a message in a global C character string. FORTRAN programs can access this error message by invoking function **PFGERR**. The message contains the name of the function in which the error occurred thus eliminating the need for a cross reference document on error codes. In this way applications programs can check for error conditions themselves and decide in what manner to use the PDBLib error messages instead of having error messages printed by the system routines. Error messages are not stacked and must be processed by the application before any other PDBLib calls are made in order to avoid potential overwrites. See the FORTRAN API section for more information about which routines return error messages

5.1 Some Guidance in Using the FORTRAN API to PDBLib

PDBLib offers a great deal of flexibility to its users. In some cases the flexibility results in a bewildering assortment of function calls from which the application developer must choose. This section is intended to offer some guidance and enable developers to home in on the functions they need to use.

5.1.1 Opening, Creating and Closing PDB Files

These are the most fundamental operations involving PDBLib. The function **PFOPEN** is used to either open an existing file or create a new one. **PFCLOS** is used to close a PDB file so that it can be recognized by PDBLib for future operations.

PDBLib allows applications to specify the binary format in which a newly created file will be written. **PFTRGT** does this work. It is not necessary to invoke **PFTRGT** before creating a new PDB file.

5.1.2 Writing Data to PDB Files

The following paragraphs describe a sequence of increasingly more elaborate output operations for PDB files. They are all compatible with one another so users can select the ones which match their needs best. The most straightforward operations are first.

There are two forms for the most basic data writing operations. These have to do with how the application wants to handle the dimension specifications. The two functions are: **PFWRTA** and **PFWRTD**.

PDBLib supports the notion of writing data of one type out into a file as another type. More precisely, an integer type of data can be written to a file in the format of any other integer type, and similarly for floating point types. The application must take all responsibility for ensuring the appropriateness of this type of conversion (e.g. underflows and overflows). The functions which support this are **PFWRAS** and **PFWRAD**.

PDBLib allows applications to append data to existing entries. This is handy in situations where the total amount of data is not known in advance, but a logical ordering of the data is apparent which matches the order in which data will be written. The functions which do this are **PFAPPA**, **PFAPPD**, **PFAPAS** and **PFAPAD**.

Finally, PDBLib allows applications to reserve space on disk and then let subsequent writes fill in that space with values. This is handy in instances where an application knows a logical structure for a data set but needs to write it out in smaller pieces and not necessarily in the order implied by its logical structure. The functions which let applications reserve space are **PFDEFA**, and **PFDEFD**. Reserved spaces may be written to with **PFWRTA**, **PFWRTD**, **PFWRAS** or **PFWRAD**.

5.1.3 Reading Data from PDB Files

Since data in a file has a definite size and shape, the reading operations in PDBLib are somewhat simpler than the writing operations. The functions **PFREAD**, **PFPTRD**, **PFRDAS**, and **PFRDAD**, the counterparts of **PFWRTA**, **PFWRTD**, **PFWRAS**, and **PFWRAD**, do all of the work in reading data from files.

5.1.4 Defining New Data Types

To aid application developers in using structured data PDBLib lets applications define new data types in a PDBfile. New data types may either be derived from other existing data types or they may be primitive (integer types, floating point types, or byte sequences). To define derived types applications should use **PFDEFS**. If a data type is already defined in an existing file, it may be copied to a new file using **PFCTYP**.

5.1.5 Defining Attributes

PDBLib supports a general mechanism for managing a class of data which is variously referred to as attributes or meta data. In a great many cases, the careful design of data structures obviates the need for this kind of data. Nevertheless, PDBLib supplies four functions to manage attributive data: **PFDATT**, **PFRATT**, **PFSVAT**, and **PFGVAT**.

5.1.6 Queries

The FORTRAN API supports several query operations to help applications interpret the contents of a PDB file. The current functions are: **PFGBFS**, **PFGERR**, **PFGMOD**, **PFGOFF**, **PFITYP**, and **PFIVAR**.

A few definitions must be given first to keep the following summary concise. Some of these will be elaborated upon in the next section.

PDBfile: for the purposes of a program this is a collection of all the relevant information about one of these data files.See the section on Data Structures for more information.

ASCII: an array of characters

VOID: any FORTRAN data type that follows the call-by-reference convention

LAST: the value 0

ENTRY: for the purposes of a program this is a collection of all the relevant information about a variable written to a PDB file.

DEFSTR: for the purposes of a program this is a collection of all the relevant information about a data type in a PDB file.

REAL*8: an eight byte floating point number.

5.2 Function Summary

These routines form the interface between FORTRAN applications programs and the PDBLib system.

PFAPAD(fileid, nchr, name, ntype, type, space, nd, ind)

Append the data from *space*of type *type* to the entry *name* in a PDB file. In this alternate to **PFAPAS** the dimensions are specified via the array *ind*.

Input: *fileid*, an integer **PDBfile** identification number,

nchr, the number of characters in *name,*

name, an ASCII string containing the name of the variable,

ntype, the number of characters in *type*,

type, an ASCII string containing the type of the variable in *space*,

space, the variable containing the data,

nd, the number of dimensions of the entry,

ind, an array containing (start, stop, step) index range triples for each dimension.

Output: Return 1 if successful, 0 otherwise.

PFAPAS(fileid, nchr, name, ntype, type, space)

Append the data from *space*of type *type* to the entry *name* in a PDB file. Dimensions are specified in the text of *name*.

Input: *fileid*, an integer **PDBfile** identification number,

nchr, the number of characters in *name,*

name, an ASCII string containing the name of the variable and any dimensional information,

ntype, the number of characters in *type*,

type, an ASCII string containing the type of the variable in *space*,

space,the variable containing the data.

Output: Return 1 if successful, 0 otherwise.

PFAPPA(fileid, nchr, name, space)

Append data to an entry in a PDB file. In this alternate to **PFAPPD** dimensions are specified in the text of *name*.

Input: *fileid*, an integer **PDBfile** identification number,

nchr, the number of characters in *name,*

name, an ASCII string containing the name of the variable and any dimensional information,

space, the variable containing the data.

Output: Return 1 if successful, 0 otherwise.

PFAPPD(fileid, nchr, name, space, nd, ind)

Append data to an entry in a PDB file. The dimensions are specified via the array *ind*.

Input: *fileid*, an integer **PDBfile** identification number,

nchr, the number of characters in *name,*

name, an ASCII string containing the name of the variable,

space, the variable containing the data,

nd, the number of dimensions of the entry,

ind, an array containing (start, stop, step) index range triples for each dimension.

Output: Return 1 if successful, 0 otherwise.

[**PFCD(fileid, nchr, dirname)**

Change the current directory in a PDB file.

Input: *fileid*, an integer **PDBfile** identification number,

nchr, the number of characters in *dirname,*

dirname, an ASCII string containing the path name of the directory to change to.

Output: Return 1 if successful, 0 otherwise.

PFCLOS(fileid)

Complete and close the PDB file *fileid*.

Input: *fileid*, an integer **PDBfile** identification number.

Output: Return 1 if successful, 0 otherwise.

PFCTYP(sfileid, dfileid, ntype, type)

Copy data type *type* from the source PDB file *sfileid* to the destination PDB file *dfileid*.

Input: *sfileid*, an integer **PDBfile** identification number for the source file,

dfileid, an integer **PDBfile** identification number for the destination file,

ntype, the number of characters in *type*,

type, an ASCII string containing the name of the type.

Output: Return 1 if successful, 0 otherwise,

PFDATT(fileid, na, attr, nt, type)

Define an attribute in a PDB file.

Input: *fileid*, an integer **PDBfile** identification number,

na, an integer number of characters in *attr*,

attr, an ASCII string naming the attribute,

nt, an integer number of characters in *type*,

type, an ASCII string specifying the attribute type.

Output: Return 1 if successful, 0 otherwise.

PFDEFA(fileid, nchr, name, ntype, outtype)

Define an entry in the symbol table of a PDB file. This function reserves space on disk but writes no data. The data can be written with later PDBLib calls.

Input: *fileid*, an integer **PDBfile** identification number,

nchr, the number of characters in *name,*

name, an ASCII string containing the name of the variable and any dimensional information,

ntype, the number of characters in *outtype*,

outtype, an ASCII string specifying the type of data in the file.

Output: Return 1 if successful, 0 otherwise.

PFDEFD(fileid, nchr, name, ntype, outtype, nd, ind)

Define an entry in the symbol table of a PDB file. This function reserves space on disk but writes no data. The data can be written with later calls. In this alternate to **PFDEFA** the dimensions are specified via the array *ind*.

Input: *fileid*, an integer **PDBfile** identification number,

nchr, the number of characters in *name,*

name, an ASCII string containing the name of the variable (no dimensional information can be supplied with the name),

ntype, the number of characters in *outtype*,

outtype, an ASCII string specifying the type of data in the file,

nd, the number of dimensions of the entry,

ind, an array containing (min, max) index range pairs for each dimension.

Output: Return 1 if successful, 0 otherwise.

PFDEFS(fileid, nchr, name, $\mathtt{ncm}_1$, $\mathtt{mem}_1$, ..., $\mathtt{ncm}_n$, $\mathtt{mem}_n$, LAST)

Define a data structure for a PDB file. **LAST** must terminate the list of members.

Input: *fileid*, an integer **PDBfile** identification number,

nchr, the number of characters in *name,*

name, an ASCII string containing the structure name,

ncm_i, the number of characters in mem_i,

mem_i, an ASCII string containing the member declaration.

Output: Return 1 if successful, 0 otherwise.

PFDEFT(fileid, nchr, name, nm, nc, memb)

Define a data structure for a PDB file.

Input: *fileid*, an integer **PDBfile** identification number,

nchr, the number of characters in *name,*

name, an ASCII string containing the structure name,

nm, the number of members,

nc, the array of offset, length pairs,

memb, an ASCII string containing the member declarations.

Output: Return 1 if successful, 0 otherwise.

PFDLS()

Release table create by **PFLST**.

Input: None.

Output: Return 1 if successful, 0 otherwise.

PFDVAR()

Release table create by **PFVART**.

Input: None.

Output: Return 1 if successful, 0 otherwise.

PFFAMI(fileid, flag)

If the file indicated by *fileid* has exceeded its size limit, open and return a new member of the file family. If *flag* is TRUE close the old file.

Input: *fileid*, an integer file identifier

flag, an integer flag.

Output: an integer file identifier

PFFLSH(fileid)

Flush the PDB file *fileid*.

Input: *fileid*, an integer **PDBfile** identification number.

Output: Return 1 if successful, 0 otherwise.

PFGBFS()

Get the current buffer size which PDBLib uses for all PDB files and return it.

Input: None.

Output: Return the current buffer size in bytes, if previously set; otherwise, return -1.

PFGERR(nchr, err)

Get the current PDBLib error message. Return the length of the error message in *nchr* and the actual message in *err*. The space for *err* should be 255 characters long.

Input: None.

Output: Return 1 if successful, 0 otherwise,

nchr, the length of the error message,

err, an ASCII string into which the message is copied.

PFGLS(n, nchr, name)

Get the name of the n^{th} entry in the internal table generated by **PFLST**.

Input: *n*, an integer ordinal index into the internal table.

Output: Return 1 if successful, 0 otherwise,

nchr, the number of characters returned in *name*,

name, an ASCII string to contain the entry name.

PFGMOD(fileid)

Get the current mode of PDB file *fileid*: 2 (append - ’a’), 3 (open - ’r’), 4 (create - ’w’).

Input: *fileid*, an integer **PDBfile** identification number.

Output: Return the integer value of the current mode.

PFGMXS(fileid)

Get the current maximum file size of PDB file *fileid*.

Input: *fileid*, an integer **PDBfile** identification number.

Output: Return the integer value of the current maximum file size.

PFGNM(fileid, nchr, name)

Return the name of a PDB file.

Input: *fileid*, an integer **PDBfile** identification number,
nchr, the number of characters in *name,*
name, an ASCII string to contain the file name.

Output: Return 1 if successful, 0 otherwise,
name, an ASCII string containing the file name.

PFGOFF(fileid)

Get the current default offset for PDB file *fileid*.

Input: *fileid*, an integer **PDBfile** identification number.

Output: Return the integer value of the default offset.

PFGVAR(n, nchr, name)

Get the name of the n^{th} variable in the internal sorted table generated by **PFVART**.

Input: *n*, integer ordinal index into the internal sorted name table.

Output: Return 1 if successful, 0 otherwise,
nchr, the number of characters in the name,
name, an ASCII array in which the name is returned.

PFGVAT(fileid, nv, var, na, attr, value)

Get the value of an attribute for an entity in a PDB file.

Input: *fileid*, an integer **PDBfile** identification number,
nv, an integer number of characters in *var*,
var, an ASCII string naming an entity,
na, an integer number of characters in *attr*,
attr, an ASCII string naming an existing attribute,
value, the data for the attribute value.

Output: Return 1 if successful, 0 otherwise.

PFIMBR(file, ntype, type, n, size, space)

Inquire about the n^{th} member of the type *type* in PDB file *fileid*.

Input: *fileid*, an integer **PDBfile** identification number,
ntype, the number of characters in *type*,
type, an ASCII string containing the name of the type,
n, an integer specifying the member,
size, an integer character size of the *space* buffer,
space, an character buffer to hold the member description.

Output: Return 1 if successful, 0 otherwise,
size, the number of characters in the member description,
space, the member description.

PFITYP(fileid, ntype, type, size, align, ind)

Inquire about type *type* in PDB file *fileid*.

Input: *fileid*, an integer **PDBfile** identification number,
ntype, the number of characters in *type*,
type, an ASCII string containing the name of the type.

Output: Return 1 if successful, 0 otherwise,
size, the number of bytes necessary to represent the type,
align, the alignment of *type* in bytes,
ind, the number of members which are pointers if *type* is a derived type.

PFIVAR(fileid, nchr, name, ntype, type, size, ndims, dims)

Inquire about variable *name* in PDB file *fileid*.

Input: *fileid*, an integer **PDBfile** identification number,
nchr, the number of characters in *name*,
name, an ASCII string containing the name of the variable.

Output: Return 1 if successful, 0 otherwise,
ntype, the number of characters in *type*,
type, an ASCII string containing the type of the variable,
size, the number of elements in the variable,
ndims, the number of dimensions which the variable has,
dims, an array of the dimensions given as (min, max) pairs.

PFLN(fileid, nvar, var, nlink, link)

Create a link to a variable in a PDB file.

Input: *fileid*, an integer **PDBfile** identification number,
nvar, the number of characters in *var,*
var, an ASCII string containing the path name of an existing variable,
nlink, the number of characters in *link,*
link, an ASCII string containing the path name of the new link.

Output: Return 1 if successful, 0 otherwise.

PFLST(fileid, npath, path, ntype, type, num)

Generate an internal table of variables and/or directories in a PDB file.

Input: *fileid*, an integer **PDBfile** identification number,
npath, the number of characters in *path,*
path, an ASCII string containing the directory and/or variable name pattern,
ntype, the number of characters in *type,*
type, an ASCII string containing the type of entries to return.

Output: Return 1 if successful, 0 otherwise,
num, the number of entries in the table.

PFMKDR(fileid, nchr, dirname)

Create a new directory in a PDB file.

Input: *fileid*, an integer **PDBfile** identification number,
nchr, the number of characters in *dirname,*
dirname, an ASCII string containing the path name of the new directory.

Output: Return 1 if successful, 0 otherwise.

PFOPEN(nchr, name, mode)

Open a PDB file on disk named *name* in one of three modes: ‘w’ for create, ‘r’ for read-only, or ‘a’ for append.

Input: *nchr*, the number of characters in *name*,

name, an ASCII string naming the file,

mode, an ASCII string, either ‘w’ for create, ‘r’ for read, or ‘a’ for append.

Output: Return an integer **PDBfile** identification number, if successful, and 0 otherwise.

PFPTRD(fileid, nchr, name, space, ind)

Read part of variable *name* from PDB file *fileid* and put it into *space*.

Input: *fileid*, an integer **PDBfile** identification number,

nchr, the number of characters in *name*,

name, an ASCII string containing the name of the variable in the PDB file,

space, the variable where the data is to be placed,

ind, the array of dimension index specifications (start, stop, step triplets for each dimension).

Output: Return 1 if successful, 0 otherwise.

PFPWD(fileid, nchr, dirname)

Get the current directory for a PDB file.

Input: *fileid*, an integer **PDBfile** identification number.

Output: Return 1 if successful, 0 otherwise,

nchr, the number of characters returned in *dirname*,

dirname, an ASCII string to contain the path name of the current directory.

PFRATT(fileid, na, attr)

Remove an attribute from a PDB file.

Input: *fileid*, an integer **PDBfile** identification number,

na, an integer number of characters in *attr*,

attr, an ASCII string naming an existing attribute.

Output: Return 1 if successful, 0 otherwise.

PFRDAD(fileid, nchr, name, ntype, type, space, ind)

Read part of variable *name* from PDB file *fileid* as variable type *type* and put it into *space*.

Input: *fileid*, an integer **PDBfile** identification number,
nchr, the number of characters in *name*,
name, an ASCII string containing the name of the variable in the PDB file,
ntype, the number of characters in *type*,
type, an ASCII string containing the type of the variable in *space*,
space, the variable where the data is to be placed,
ind, the array of dimension index specifications (start, stop, step triplets for each dimension).

Output: Return 1 if successful, 0 otherwise.

PFRDAS(fileid, nchr, name, ntype, type, space)

Read variable *name* from PDB file *fileid* as variable type *type* and put it into *space*.

Input: *fileid*, an integer **PDBfile** identification number,
nchr, the number of characters in *name*,
name, an ASCII string containing the name of the variable in the PDB file,
ntype, the number of characters in *type*,
type, an ASCII string containing the type of the variable in *space*,
space, the variable where the data is to be placed.

Output: Return 1 if successful, 0 otherwise.

PFREAD(fileid, nchr, name, space)

Read variable *name* from PDB file *fileid* and put it into *space*.

Input: *fileid*, an integer **PDBfile** identification number,
nchr, the number of characters in *name*,
name, an ASCII string containing the name of the variable in the PDB file,
space, the variable where the data is to be placed.

Output: Return 1 if successful, 0 otherwise.

PFSBFS(v)

Set the buffer size which PDBLib will use for all PDB files to *v*.

Input: *v*, an integer value for the buffer size in bytes.

Output: Return the integer value of the buffer size in bytes.

PFSMXS(fileid, v)

Set the maximum file size for PDB file *fileid* to *v*.

Input: *fileid*, an integer **PDBfile** identification number.

v, an integer value for the maximum file size.

Output: Return 1 if successful, 0 otherwise.

PFSOFF(fileid, v)

Set the default offset for PDB file *fileid* to *v*.

Input: *fileid*, an integer **PDBfile** identification number,

v, an integer value for the default offset.

Output: Return 1 if successful, 0 otherwise.

PFSVAT(fileid, nv, var, na, attr, value)

Set the value of an attribute for an entity in a PDB file.

Input: *fileid*, an integer **PDBfile** identification number,

nv, an integer number of characters in *var*,

var, an ASCII string naming an entity,

na, an integer number of characters in *attr*,

attr, an ASCII string naming an existing attribute,

value, the data for the attribute value.

Output: Return 1 if successful, 0 otherwise.

PFTRGT(is, ia)

Write the next PDB file to be created according to the specified data standard and alignment.

Input: *is*, an index specifying the data standard,

ia, an index specifying the data alignment.

Output: Return 1 if successful, 0 otherwise.

PFVART(fileid, order, nvars)

Generate an internal table of variables in PDB file *fileid* which is sorted according to the specification, *order*.

Input: *fileid*, an integer **PDBfile** identification number,

order, an integer specifying the order (1, alphabetic; 2, disk address order).

Output: Return 1 if successful, 0 otherwise,

nvars, the number of variables in the file.

PFWIMA(fileid, nchr, name, pkn, pkx, pln, plx, data, pxn, pxx, pyn, pyx, pim)

Write the given information out to file *fileid* as a **PD_image**.

Input: *fileid*, an integer **PDBfile** identification number,
nchr, the number of characters in the image name,
name, an ASCII string containing the image name,
pkn, an integer containing the minimum column index of *data*,
pkx, an integer containing the maximum column index of *data*,
pln, an integer containing the minimum row index of *data*,
plx, an integer containing the maximum row index of *data*,
data, a real*8 array containing the range data,
pxn, a real*8 constant specifying the minimum column index in image,
pxx, a real*8 constant specifying the maximum column index in image,
pyn, a real*8 constant specifying the minimum row index in image,
pyx, a real*8 constant specifying the maximum row index in image,
pim, an integer containing the image number.

Output: Return 1 if successful, 0 otherwise.

PFWMAP(fileid, dname, dp, dm, rname, rp, rm, pim)

Write the given information out to file *fileid* as a **PM_mapping**. The number *pim* represents the mapping number and must be unique.

Input: *fileid*, an integer **PDBfile** identification number,
dname, an ASCII string containing the domain name,
dp, an integer array of parameters defining the domain,
dm, a real*8 array containing the domain data,
rname, an ASCII string containing the range name,
rp, an integer array of parameters defining the range,
rm, a real*8 array containing the range data,
pim, an integer containing the mapping number.

Output: Return 1 if successful, 0 otherwise.

PFWRAD(fileid, nchr, name, ntypin, intype, ntypout, outtype, space, nd, ind)

Write the data pointed to by *space* under *name* and with *type* in PDB file *fileid*. Array *ind* contains *nd* triples indicating the minimum and maximum index and the stride for each dimension.

Input: *fileid*, an integer **PDBfile** identification number,

nchr, the number of characters in *name*,

name, an ASCII string containing the name of the variable in the PDB file,

ntypin, the number of characters in *intype*,

intype, an ASCII string containing the type of the variable in *space*,

ntypout, the number of characters in *outtype*,

outtype, an ASCII string containing the type of the variable in *name*,

space, the variable containing the data,

nd, an integer number of dimensions,

ind, an array of *nd* integer (min, max, stride) triples.

Output: Return 1 if successful, 0 otherwise.AS(fileid, nchr, name, ntypin, intype, ntypout, outtype, space)

PFWRAN(fileid, dname, nchr, rname, rp, rm, pim)

Write the given information out to file *fileid* as a **PM_mapping**. The number *pim* represents the mapping number and must be unique.

Input: *fileid*, an integer **PDBfile** identification number,

dname, an ASCII string containing the domain name,

nchr, the number of characters in the domain name,

rname, an ASCII string containing the range name,

rp, an integer array of parameters defining the range,

rm, a real*8 array containing the range data,

pim, an integer containing the mapping number.

Output: Return 1 if successful, 0 otherwise.

PFWRAS(fileid, nchr, name, ntypin, intype, ntypout, outtype, space)

Write the data pointed to by *space* of type *intype* in PDB file *fileid* under *name* and with type *outtype*. An arbitrary number of dimensions may be included in the *name* string.

Input: *fileid*, an integer **PDBfile** identification number,

nchr, the number of characters in *name*,

name, an ASCII string containing the name of the variable in the PDB file,

ntypin, the number of characters in *intype*,

intype, an ASCII string containing the type of the variable in *space*,

ntypout, the number of characters in *outtype*,

outtype, an ASCII string containing the type of the variable in *name*,

space, the variable containing the data.

Output: Return 1 if successful, 0 otherwise.

PFWRTA(fileid, nchr, name, ntype, type, space)

Write the data pointed to by *space* under *name* and with *type* in PDB file *fileid*. An arbitrary number of dimensions may be included in the *name* string.

Input: *fileid*, an integer **PDBfile** identification number,

nchr, the number of characters in *name*,

name, an ASCII string containing the name of the variable in the PDB file,

ntype, the number of characters in *type*,

type, an ASCII string containing the type of the variable,

space, the variable containing the data.

Output: Return 1 if successful, 0 otherwise.

PFWRTD(fileid, nchr, name, ntype, type, space, nd, ind)

Write the data pointed to by *space* under *name* and with *type* in PDB file *fileid*. Array *ind* contains *nd* triples indicating the minimum and maximum index and the stride for each dimension.

Input: *fileid*, an integer **PDBfile** identification number,

nchr, the number of characters in *name*,

name, an ASCII string containing the name of the variable in the PDB file,

ntype, the number of characters in *type*,

type, an ASCII string containing the type of the variable,

space, the variable containing the data,

nd, an integer number of dimensions,

ind, an array of *nd* integer (min, max, stride) triples.

Output: Return 1 if successful, 0 otherwise.

PFWSET(fileid, dname, dp, dm)

Build a **PM_set** structure out of the given input data and write it to a PDB file.

Input: *fileid*, an integer **PDBfile** identification number,
dname, an ASCII string containing the set name,
dp, an integer array of parameters defining the **PM_set**,
dm, an array of real*8 values containing the set elements.

Output: Return 1 if successful, 0 otherwise.

6.0 The FORTRAN API

In this section a more formal description of the FORTRAN API for PDBLib is given including a more detailed account of their workings.

6.1 PFAPAD — (PD_append_as_alt)

```
integer PFAPAD(integer fileid,
               integer nchr,
               character name,
               integer ntype,
               character intype,
               void space,
               integer nd,
               integer ind)
```

Append data of type *intype* to an entry in the specified file. The outtype is assumed to be the same as for the original entry. The dimensions of the appended data are specified in *nd* and *ind*. They must match the original entry except for the most slowly varying one. The specification of the most slowly varying dimension must be one of the following;

min:max:stride => new dimension is old_min:old_max+(max-min+1)

if min is the default_offset for the file

or

min:max:stride => new dimension is old_min:max

if min is old_max+1

The rationale for this function is that some data sets are of unknown size until they are completely written. PDBLib permits any entry to reside in discontiguous blocks of disk space. The library is responsible for reading and writing data correctly across these blocks.

The shape or dimensional information of the entry is specified in *nd* and *ind*. In this respect **PFAPAD** behaves just like **PFWRTD**.

The arguments to **PFAPAD** are: *fileid*, an integer identifier which designates the PDB file to which to write; *nchr*, an integer number of characters in the *name* string; *name*, an ASCII string containing the name of the variable to install in the symbol table; *ntype*, an integer number of characters in the *intype* string; *intype*, an ASCII string specifying the type of data to which *space* points; *space*, the data to be written; *nd*, an integer number of dimensions; and *ind*, an array of *nd* integer triples containing the minimum and maximum index and the stride for each dimension.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFAPAS**, **PFAPPA**, **PFAPPD**, **PFWRAD**, **PFWRAS**, **PFWRTA**, and **PFWRTD**.

```
integer PFAPAD(integer fileid,
               integer nchr,
               character name,
               integer ntype,
               character intype,
               void space,
               integer nd,
               integer ind)
```

Example

```
      character*8 intype
      integer pfapad, pfwrtd
      integer fileid, ntype, nd, ind(3)
      real x(20)
      real*8 y(10)
            .
            .
            .
c     write array x
      ind(1) = 1
      ind(2) = 20
      ind(3) = 1
      if (pfwrtd(fileid, 1, ‘x’, 5, ‘float’, x, 1, ind) .eq. 0)
     $     call errproc
            .
            .
            .
c     append to x
      ind(2) = 10
      if (pfapad(fileid, 1, ‘x’, 6, ‘double’, y, 1, ind) .eq. 0)
     $     call errproc
            .
            .
            .
```

Compare this with the example of **PFWRAD**.

6.2 PFAPAS — (PD_append_as)

```
integer PFAPAS(integer fileid,
               integer nchr,
               character name,
               integer ntype,
               character intype,
               void space)
```

Append data to an entry in the specified file. The outtype is assumed to be the same as for the original entry. The dimensions of the appended data are specified in *name*. They must match the original entry except for the most slowly varying one. The specification of the most slowly varying dimension must be one of the following;

min:max => new dimension is old_min:old_max+(max-min+1)

if min is the default_offset for the file

or

min:max => new dimension is old_min:max

if min is old_max+1

The rationale for this function is that some data sets are of unknown size until they are completely written. PDBLib permits any entry to reside in discontiguous blocks of disk space. The library is responsible for reading and writing data correctly across these blocks.

The shape or dimensional information of the entry is a part of the name string. In this respect **PFAPAS** behaves just like **PFWRAS**.

The arguments to **PFAPAS** are: *fileid*, an integer identifier which designates the PDB file to which to write; *nchr*, an integer number of characters in the *name* string; *name*, an ASCII string containing the name of the variable and any dimensional information, *ntype*, an integer number of characters in the *intype* string; *intype*, an ASCII string specifying the type of data to which *space* points and *space*, the data to be appended.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFAPAD**, **PFAPPA**, **PFAPPD**, **PFWRAD**, **PFWRAS**, **PFWRTA**, and **PFWRTD**.

```
integer PFAPAS(integer fileid,
               integer nchr,
               character name,
               integer ntype,
               character intype,
               void space)
```

Example

```
      character*8 intype
      integer pfapas, pfwrta
      integer fileid, nchr, ntype
      real x(20)
      real*8 y(10)
             .
             .
             .
c     write array x
      if (pfwrta(fileid, 5, 'x(20)', 5, 'float', x) .eq. 0)
     $     call errproc
             .
             .
             .
c     append to x
      if (pfapas(fileid, 7, 'x(1:10)', 6, 'double', y) .eq. 0)
     $     call errproc
             .
             .
             .
```

Compare this with the example of **PFWRAS**.

6.3 PFAPPA — (PD_append)

```
integer PFAPPA(integer fileid,
                  integer nchr,
                  character name,
                  void space)
```

Append data to an entry in the specified file. The type is assumed to be the same as for the original entry. The dimensions of the appended data are specified in *name*. They must match the original entry except for the most slowly varying one. The specification of the most slowly varying dimension must be one of the following;

min:max => new dimension is old_min:old_max+(max-min+1)
if min is the default_offset for the file

or

min:max => new dimension is old_min:max
if min is old_max+1

The rationale for this function is that some data sets are of unknown size until they are completely written. PDBLib permits any entry to reside in discontiguous blocks of disk space. The library is responsible for reading and writing data correctly across these blocks.

The shape or dimensional information of the entry is a part of the name string. In this respect **PFAPPA** behaves just like **PFWRTA**.

The arguments to **PFAPPA** are: *fileid*, an integer identifier which designates the PDB file to which to write; *nchr*, an integer number of characters in the *name* string; *name*, an ASCII string containing the name of the variable to install in the symbol table; and *space*, the data to be written.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFAPAD**, **PFAPAS**, **PFAPPD**, **PFWRAD**, **PFWRAS**, **PFWRTA**, and **PFWRTD**.

```
integer PFAPPA(integer fileid,
               integer nchr,
               character name,
               void space)
```

Example

```
      integer pfappa, pfwrta
      integer fileid
      real x(20), y(10)
             .
             .
             .
c     write array x
      if (pfwrta(fileid, 5, 'x(20)', 5, 'float', x) .eq. 0)
     $     call errproc

c     append to x
      if (pfappa(fileid, 7, 'x(1:10)', y) .eq. 0)
     $     call errproc
             .
             .
             .
```

6.4 PFAPPD — (PD_append_alt)

```
integer PFAPPD(integer fileid,
               integer nchr,
               character name,
               void space,
               integer nd,
               integer ind)
```

Append data to an entry in the specified file. The type is assumed to be the same as for the original entry. The dimensions of the appended data are specified in *nd* and *ind*. They must match the original entry except for the most slowly varying one. The specification of the most slowly varying dimension must be one of the following;

min:max:stride => new dimension is old_min:old_max+(max-min+1)

if min is the default_offset for the file

or

min:max:stride => new dimension is old_min:max

if min is old_max+1

The rationale for this function is that some data sets are of unknown size until they are completely written. PDBLib permits any entry to reside in discontiguous blocks of disk space. The library is responsible for reading and writing data correctly across these blocks.

The shape or dimensional information of the entry is specified in *nd* and *ind*. In this respect **PFAPPD** behaves just like **PFWRTD**.

The arguments to **PFAPPD** are: *fileid*, an integer identifier which designates the PDB file to which to write; *nchr*, an integer number of characters in the *name* string; *name*, an ASCII string containing the name of the variable to install in the symbol table; *space*, the data to be written; *nd*, an integer number of dimensions; and *ind*, an array of *nd* integer triples containing the minimum and maximum index and the stride for each dimension.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFAPAD**, **PFAPAS**, **PFAPPA**, **PFWRAD**, **PFWRAS**, **PFWRTA**, and **PFWRTD**.

```
integer PFAPPD(integer fileid,
               integer nchr,
               character name,
               void space,
               integer nd,
               integer ind)
```

Example

```
      integer pfappd, pfwrtd
      integer fileid, ind(3)
      real x(20), y(10)
             .
             .
             .
c     write array x
      ind(1) = 1
      ind(2) = 20
      ind(3) = 1
      if (pfwrtd(fileid, 1, 'x', 5, 'float', x, 1, ind) .eq. 0)
     $     call errproc

c     append to x
      ind(2) = 10
      if (pfappd(fileid, 1, 'x', y, 1, ind) .eq. 0)
     $     call errproc
             .
             .
             .
```

6.5 PFCD — (PD_cd)

```
integer PFCD(integer fileid,
             integer nchr,
             char *dirname)
```

Change the current directory in a PDB file.

PDBLib supports an optional hierarchical directory structure inside PDB files. A directory or a variable in a directory may be specified by either a relative path or an absolute path. Slashes separate nodes in a path name. Absolute paths begin with a slash. Nodes consisting of two periods, "..", refer to the next higher level directory.

The arguments to **PFCD** are: *fileid*, an integer file identifier; *nchr*, an integer number of characters in string *dirname*; and *dirname*, an ASCII string containing the path name of the directory to change to.

If *dirname* is an empty string or a slash, it refers to the top level or root directory.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFDLS**, **PFGLS**, **PFLN**, **PFLST**, **PFMKDR**, **PFPWD**.

```
integer PFCD(integer fileid,
             integer nchr,
             char *dirname)
```

Example

```
       integer pfcd
       integer fileid
             .
             .
             .
       if (pfcd(fileid, 16, '/animals/mammals') .eq. 0)
      $     call errproc
             .
             .
             .
       if (pfcd(fileid, 11, '../reptiles') .eq. 0)
      $     call errproc
             .
             .
             .
```

6.6 PFCLOS — (PD_close)

```
integer PFCLOS(integer fileid)
```

Close a PDB file. After all data is written to the PDB file, the structure chart and symbol table must be written out to the file and their disk addresses recorded in the file header. Without these operations the file cannot be read back in by PDBLib and all data is lost.

All open PDB files must be **PFCLOS**'d before exiting the program. The integer identifier *fileid* associated with the PDB file must be passed to **PFCLOS**.

This function returns 1 if the PDB file is correctly written and closed; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

```
integer PFCLOS(integer fileid)
```

Example

```
      integer pfclos
      integer fileid
             .
             .
             .
      if (pfclos(fileid) .eq. 0)
     $      call errproc
             .
             .
             .
```

6.7 PFDATT — (PD_def_attribute)

```
integer PFDATT(integer fileid,
               integer na,
               char *attr,
               integer nt,
               char *type)
```

Define an attribute to the given PDB file. The model of an attribute in PDBLib is an entity that has a name and type. The two supported operations on attributes are to create them and to remove them. An entity in a PDB file can be assigned an attribute value simply by making a call which specifies the entity name, the attribute name, and the attribute value (which is determined by the type). The only association between an entry in a PDB file and any attribute is made by the name in the attribute table and the entry in the symbol table. In particular, this mechanism allows the application developer to define and use entities in a PDB file solely in terms of attributes.

The arguments to **PFDATT** are: *fileid*, an integer identifier which designates the PDB file in which the attribute is being defined; *na*, an integer number of characters in the *attr* string; *attr*, an ASCII string containing the name of the attribute; *nt*, an integer number of characters in the *type* string; and *type*, an ASCII string containing the type of the attribute;

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFRATT**, **PFGVAT**, and **PFSVAT**.

```
integer PFDATT(integer fileid,
               integer na,
               char *attr,
               integer nt,
               char *type)
```

Example

```
      integer pfdatt
      integer fileid
             .
             .
             .
      if (pfdatt(fileid, 4, 'date', 6, 'char *') .eq. 0)
     $     call errproc
             .
             .
             .
```

6.8 PFDEFA — (PD_defent)

```
integer PFDEFA(integer fileid,
               integer nc,
               char *name,
               integer nt,
               char *type)
```

Define an entry in the symbol table of the PDB file specified by *fileid*. This function reserves space on disk but writes no data. The data can be written with later calls to **PFWRTA** or **PFWRTD**.

The shape or dimensional information of the entry is a part of the *name* string. In this respect it behaves as **PFWRTA**.

The rationale for this function is to block out space in a PDB file corresponding to some logical layout of a piece of data. The data may not exist at the time the space is reserved or for some reason it may be desirable to write out the data in pieces. In any case if the type and shape of a variable is known at some point, an entry may be made in the PDB file without writing any data. The space may filled with other PDBLib calls at some later time.

The arguments to this function are: *fileid*, an integer identifier which designates the PDB file in which to define an entry; *nchr*, an integer number of characters in the *name* string; *name*, an ASCII string containing the name of the variable to install in the symbol table; *ntype*, an integer number of characters in the *type* string; and *type*, an ASCII string specifying the variable type.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFAPAD**, **PFAPAS**, **PFAPPA**, **PFAPPD**, **PFDEFD**, **PFWRAD**, **PFWRAS**, **PFWRTA**, and **PFWRTD**.

```
integer PFDEFA(integer fileid,
               integer nc,
               char *name,
               integer nt,
               char *type)
```

Example

```
      integer pfdefa
      integer fileid
            .
            .
            .
c     define and reserve array x
c     declaration for x would be: real x(20)

      if (pfdefa(fileid, 5, 'x(20)', 5, 'float') .eq. 0)
     $     call errproc
            .
            .
            .
```

Compare this with the example of **PFWRTA**.

6.9 PFDEFD — (PD_defent_alt)

```
integer PFDEFD(integer fileid,
               integer nc,
               char *name,
               integer nt,
               char *type,
               integer nd,
               integer ind)
```

Define an entry in the symbol table of the PDB file specified by *fileid*. This function reserves space on disk but writes no data. The data can be written with later calls to **PFWRTA** or **PFWRTD**.

This is an alternate form of **PFDEFA**. The difference is that the dimension information is supplied via the *nd* and *ind* arguments instead of being a part of the name string. In this respect it behaves as **PFWRTD**.

The rationale for this function is to block out space in a PDB file corresponding to some logical layout of a piece of data. The data may not exist at the time the space is reserved or for some reason it may be desirable to write out the data in pieces. In any case if the type and shape of a variable is known at some point, an entry may be made in the PDB file without writing any data. The space may filled with other PDBLib calls at some later time.

The arguments to this function are: *fileid*, an integer identifier which designates the PDB file in which to define an entry; *nchr*, an integer number of characters in the *name* string; *name*, an ASCII string containing the name of the variable to install in the symbol table; *ntype*, an integer number of characters in the *type* string; *type*, an ASCII string specifying the variable type; *nd*, an integer number of dimensions; and *ind*, an array of *nd* integer pairs containing the minimum and maximum index for each dimension.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFAPAD**, **PFAPAS**, **PFAPPA**, **PFAPPD**, **PFDEFA**, **PFWRAD**, **PFWRAS**, **PFWRTA**, and **PFWRTD**.

See also **PFAPPA**, **PFAPPD**, **PFDEFA**, **PFWRTA**, and **PFWRTD**.

```
integer PFDEFD(integer fileid,
               integer nc,
               char *name,
               integer nt,
               char *type,
               integer nd,
               integer ind)
```

Example

```
      integer pfdefd
      integer fileid, nd, ind(4)
             .
             .
             .
c     define and reserve array c
c     declaration for c would be: real c(2,2:4)

      nd = 2
      ind(1) = 1
      ind(2) = 2
      ind(3) = 2
      ind(4) = 4
      if (pfdefd(fileid, 1, 'c', 5, 'float', nd, ind) .eq. 0)
     $     call errproc
             .
             .
             .
```

Compare with the example for **PFWRTD**.

6.10 PFDEFS — (PD_defstr))

```
integer PFDEFS(integer fileid,
               integer nchr,
               character name,
               integer ncm1, character mem1,
                    ...,
               integer ncmn, character memn,
               integer LAST)
```

Define a data structure for a PDB file. As a matter of programming efficiency and clarity it is useful to be able to refer to more complex structural units than the primitive types: short integers, integers, long integers, floating point numbers, double precision floating point numbers, and characters. Arrays do this in a very simple-minded way. Many modern languages support extended types or structures which allow the programmer to group diverse types of data together in a very sophisticated way.

PDBLib supports an extended data typing mechanism called a structure. A structure is a set of declarations of members. Each member is in turn a data type known to the system. Much of the style and usage of structures comes from the C struct. **Note:** because FORTRAN 77 lacks a pointer type, structures defined with this function should not contain pointered members.

PFDEFS defines data structures to the PDB system so that instances of such extended types can be read and written as a whole in a single statement. The members of the structure are processed and an entry in the structure chart is made. Subsequent references to the new structure type are processed using information from the structure chart. The syntax by which members of a structure are specified is like that for C structs. The formal definition is given below ([ ] enclose optional elements). **Note:** self-referential structures should not be used in FORTRAN applications.

<member>	:=	<type> <member name>[(<dimensions>)]
<type>	:=	<primitive type> | <derived type>
<member name>	:=	an ASCII string representing the name of the member
<primitive type>	:=	short | integer | long | float | double | char
<derived type>	:=	any **PFDEFS**'d type
<dimensions>	:=	<integer> |
		<integer : integer> |
		<integer>, <dimensions> |
		<integer : integer>, <dimensions> |

Dimensions can be given in two ways. If a single number is specified for a given dimension, it is interpreted as the length of that dimension and the default offset value for the PDB file is taken as the minimum value. In this case the maximum value is the default offset plus the minimum value minus one. Alternatively, the minimum value followed by a colon and the maximum value may be specified. For example,

```
integer a(30,1:10)
```

The arguments to **PFDEFS** are: *fileid*, an integer identifier which designates the PDB file to which to write; *nchr*, the integer number of characters in the *name* string; *name*, an ASCII string containing the name of the variable to install in the symbol table; a list of pairs where ncm_i is an integer number of characters in the mem_i string which follows it; and a terminating **LAST**.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

```
integer PFDEFS(integer fileid,
               integer nchr,
               character name,
               integer ncm1, character mem1,
                    ...,
               integer ncmn, character memn,
               integer LAST)
```

Example

```
      parameter(LAST = 0)
      integer pfdefs
      integer fileid
      common /abc/ a(2), b, c(2,2:4)
      real a, b, c
      common /jkl/ j, k, l
      integer j, k, l
            .
            .
            .
      if (pfdefs(fileid,
     $           3,  'abc',
     $           10, 'float a(2)',
     $           7,  'float b',
     $           14, 'float c(2,2:4)',
     $           LAST) .eq. 0)
     $     call errproc

      if (pfdefs(fileid,
     $           3, 'jkl',
     $           9, 'integer j',
     $           9, 'integer k',
     $           9, 'integer l',
     $           LAST) .eq. 0)
     $     call errproc
            .
            .
            .
```

6.11 PFDEFT — (PD_defstr_alt)

```
integer PFDEFT(integer fileid,
               integer nchr,
               character name,
               integer nm,
               integer nc(2*nm),
               character *memb)
```

Define a data structure for a PDB file. This is an alternate form of PFDEFS (see the description above for a discussion of structured data types).

The arguments to **PFDEFT** are: *fileid*, an integer identifier which designates the PDB file to which to write; *nchr*, the integer number of characters in the *name* string; *name*, an ASCII string containing the name of the variable to install in the symbol table; *nm*, an integer specifying how many members the type has; *nc*, an array of integer pairs specifying the 0 based offset into the *nm* array and the number of characters for each member; and *nm*, an array of characters containing the member specifications.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFDEFS**.

```
integer PFDEFT(integer fileid,
               integer nchr,
               character name,
               integer nm,
               integer nc(2*nm),
               character *memb)
```

Example

```
integer pfdeft
integer fileid, nc(6)
character*8 nm(4)
common /abc/ a(2), b, c(2,2:4)
      .
      .
      .
nc(1) = 0
nc(2) = 10
nc(3) = 10
nc(4) = 7
nc(5) = 17
nc(6) = 14
nm(1) = ‘float a(‘
nm(2) = ‘2)float ‘
nm(3) = ‘bfloat c’
nm(4) = ‘(2,2:4) ‘
if (pfdeft(fileid, 3, ’abc’, 3, nc, nm) .eq. 0)
   call errproc
      .
      .
      .
```

Compare with the example for **PFDEFS**.

6.12 PFFAMI — (PD_family)

```
integer PFFAMI(integer fileid, integer flag)
```

This function checks to see whether the specified file has exceeded it size limit. If it has a new file is opened and returns. If not the given file pointer is returned. The flag is set to TRUE if you want **PFFAMI** to close the file it is given. Otherwise the application is responsible for closing the file

The arguments to this function are: *fileid*, an integer file identifier returned by a previous call to **PFOPEN**; and *flag* an integer value (either TRUE or FALSE).

This function returns an integer file identifier.

```
integer PFFAMI(integer fileid, integer flag)
```

Example

```
integer old, new
integer pffami
        .
        .
        .
 new = pffami(old, 0)
 if (new .ne. old) then
    pfclos(old)
          .
          .
          .
    old = new
 endif
        .
        .
        .
```

6.13 PFFLSH — (PD_flush)

```
integer PFFLSH(integer fileid)
```

This function writes out the information which describes the contents of the PDB file specified. Normally, **PFCLOS** calls this routine, but applications that want to protect themselves from system failures or other problems may chose to periodically use this function. After a successful return and until or unless more data is written to the file or space reserved for future writes, the PDB file is valid in the sense that if the application terminates unexpectedly before calling **PFCLOS**, the file can be **PFOPEN**'d successfully.

NOTE: this call does NOT obviate **PFCLOS**!

The argument to this function is *file*, the pointer to the PDBfile structure returned by a previous call to **PD_open**.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

`integer PFFLSH(integer fileid)`

Example

```
      integer pfflsh
      integer fileid
             .
             .
             .
      if (pfflsh(fileid) .eq. 0)
     $     call errproc
             .
             .
             .
```

6.14 PFGFNM

```
integer PFGFNM(integer fileid,
               integer nchr,
               character name)
```

Return the name of the specified PDB file. The *nchr* argument contains the length of the buffer on input and the length of the file name on output. If the buffer is not long enough, the length of the file name is returned in *nchr* and a value of 0 is returned.

The arguments to **PFGFNM** are: *fileid*, an integer identifier which designates the PDB file in question; *nchr*, an integer number of characters in the *name* string; *name*, an ASCII string which will contain the file name upon successful completion.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

```
integer PFGFNM(integer fileid,
               integer nchr,
               character name)
```

Example

```
      integer pfgfnm
      integer fileid
      character*8 name(4)
             .
             .
             .
      if (pfgfnm(fileid, 32, name) .eq. 0)
     $     call errproc
             .
             .
             .
```

6.15 PFGLS

```
integer PFGLS(integer n,
              integer nchr,
              char *name)
```

Get the name of the n^{th} entry in the internal table generated by **PFLST**.

PDBLib supports an optional hierarchical directory structure inside PDB files. A directory or a variable in a directory may be specified by either a relative path or an absolute path. Slashes separate nodes in a path name. Absolute paths begin with a slash. Nodes consisting of two periods, "..", refer to the next higher level directory.

The arguments to **PFGLS** are: *n*, an integer ordinal index into the internal table; *nchr*, an integer to contain the number of characters returned in string *name*; and *name*, an ASCII string to contain the returned entry name.

Calls to **PFGLS** must be preceded by a call to **PFLST** to set up an internal table of directory and/or variable names.

This function copies the name of a symbol table entry into the *name*, if successful. The application must ensure that *name* is large enough to contain the longest name in the symbol table (although PDBLib permits names of arbitrary length, 256 characters would probably be more than enough).

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFCD**, **PFDLS**, **PFLN**, **PFLST**, **PFMKDR**, **PFPWD**.

```
integer PFGLS(integer n,
              integer nchr,
              char *name)
```

Example

```
      integer pfgls
      integer i, nvar, nchr
      character name(256)
             .
             .
             .
      do 100 i = 1, nvar
        if (pfgls(i, nchr, name) .eq. 0)
     $       call errproc
             .
             .
             .
  100 continue
             .
             .
             .
```

6.16 PFGVAR

```
integer PFGVAR(n, nchr, name)
```

Get the name of the n^{th} variable in the internal table generated by a previous call to **PFVART**. The table will have been sorted in a particular order and this function allows applications to access the variables in the sorted order, not the default hash ordering that would normally apply.

This function copies the name into the array *name*, if successful. The application must ensure that *name* is large enough to contain the longest name in the symbol table (although PDBLib permits names of arbitrary length, 256 characters would probably be more than enough).

The arguments to **PFGVAR** are: *n*, an integer ordinal index into the internal sorted name table; *nchr*, an integer in which the number of characters in the *name* string is returned; *name*, an ASCII string which will contain the variable name upon successful completion.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFVART** and **PFDVAR**.

```
integer PFGVAR(n, nchr, name)
```

Example

```
      integer pfgvar
      integer i, n, nvar, nchar
      character vname(256)
            .
            .
            .
c     print out the table of variable names created by pfvart
      write(6,700)
 700  format(/,'Alphabetic list of variables:')
      do 701 n = 1, nvar
         if (pfgvar(n, nchar, vname) .eq. 0)
     $        call errproc
         write(6,702) (vname(i), i=1,nchar)
 702     format('           ',256a1)
 701  continue
            .
            .
            .
```

6.17 PFGVAT — (PD_get_attribute)

```
integer PFGVAT(integer fileid,
               integer nv,
               char *var,
               integer na
               char *attr,
               void value)
```

Get the value of the specified attribute for the named entity. The application must ensure that there is sufficient space to receive the attribute value.

The model of an attribute in PDBLib is an entity that has a name and type. The two supported operations on attributes are to create them and to remove them. An entity in a PDB file can be assigned an attribute value simply by calling **PFSVAT**. The only association between an entry in a PDB file and any attribute is made by the name in the attribute table and the entry in the symbol table. In particular, this mechanism allows the application developer to define and use entities in a PDB file solely in terms of attributes.

The arguments to **PFGVAT** are: *fileid*, an integer identifier which designates the PDB file from which to get the attribute; *nv*, an integer number of characters in the *var* string; *var*, an ASCII string containing the name of an entity; *na*, an integer number of characters in the *attr* string; *attr*, an ASCII string containing the name of the attribute; and *value* into which the attribute value will be copied.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFDATT**, **PFRATT**, and **PFSVAT**.

```
integer PFGVAT(integer fileid,
               integer nv,
               char *var,
               integer na
               char *attr,
               void value)
```

Example

```
      integer pfgvat
      integer fileid
      character*8 dt(10)
             .
             .
             .
      if (pfgvat(fileid, 3, 'foo', 4, 'date', dt) .eq. 0)
     $     call errproc
             .
             .
             .
```

6.18 PFIMBR

```
integer PFIMBR(integer fileid,
               integer ntype,
               char *type,
               integer n
               integer size
               char *space)
```

Inquire about the n^{th} member of the type *type* in PDB file *fileid*. Return the description of the member in the character buffer, *space*. On input *size* is the number of characters in the buffer *space*. If the member description requires more space, *size* is set to the number of bytes required and 0 is returned.

Input to this function is: *fileid*, an integer **PDBfile** identification number*; ntype*, the number of characters in *type*; *type*, an ASCII string containing the name of the type; *n*, an integer specifying the member; *size*, an integer character size of the *space* buffer; and *space*, an character buffer to hold the member description.

Output from this function is: *size*, the number of characters in the member description; and *space*, the member description.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFDEFS**.

```
integer PFIMBR(integer fileid,
               integer ntype,
               char *type,
               integer n
               integer size
               char *space)
```

Example

```
      integer pfimbr
      integer fileid
      character*8 type(10), desc(10), bdesc(100)
             .
             .
             .
      size = 80
      if (pfimbr(fileid, 3, ‘foo’, 4, size, desc) .eq. 0) then
         if (size .le. 800) then
            if (pfimbr(fileid, 3, ‘foo’, 4, size, bdesc) .eq. 0) then
     $         call errproc
            endif
         endif
      endif
             .
             .
             .
```

6.19 PFLN — (PD_ln)

```
integer PFLN(integer fileid,
             integer nvar,
             char *var,
             integer nlink,
             char *link)
```

Create a link to a variable in a PDB file.

PDBLib supports an optional hierarchical directory structure inside PDB files. A directory or a variable in a directory may be specified by either a relative path or an absolute path. Slashes separate nodes in a path name. Absolute paths begin with a slash. Nodes consisting of two periods, "..", refer to the next higher level directory.

The arguments to **PFLN** are: *fileid*, an integer file identifier; *nvar*, an integer number of characters in string *var*; *var*, an ASCII string containing the path name of an existing variable; *nlink*, an integer number of characters in string *link*; and *link*, an ASCII string containing the path name of the new link.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFCD**, **PFDLS**, **PFGLS**, **PFLST**, **PFMKDR**, **PFPWD**.

```
integer PFLN(integer fileid,
             integer nvar,
             char *var,
             integer nlink,
             char *link)
```

Example

```
      integer pfln
      integer fileid
             .
             .
             .
      if (pfln(fileid, 27, '/animals/mammals/chimpanzee',
     $                  6, '/chimp') .eq. 0)
     $     call errproc
             .
             .
             .
```

6.20 PFLST — (PD_ls)

```
integer PFLST(integer fileid,
              integer npath,
              char *path,
              integer ntype,
              char *type,
              integer num)
```

Generate an internal table of names of entries (variables and/or directories) in a PDB file that are of a specified type and that are in the directory and match the variable name pattern specified.

PDBLib supports an optional hierarchical directory structure inside PDB files. A directory or a variable in a directory may be specified by either a relative path or an absolute path. Slashes separate nodes in a path name. Absolute paths begin with a slash. Nodes consisting of two periods, "..", refer to the next higher level directory.

The arguments to **PFLST** are: *fileid*, an integer file identifier; *npath*, an integer number of characters in string *path*; *path*, an ASCII string containing the path name of the directory to search and/or the variable name pattern to match; *ntype*, an integer number of characters in string *type*; *type*, an ASCII string containing the type of entries to return; and *num*, an integer to contain the number of entries returned.

If *npath* is zero, the contents of the current directory are listed. If *ntype* is zero, all types are returned.

The terminal node of *path* may contain meta characters "*" and "?". Each "*" matches any zero or more characters and each "?" matches any single character.

Call **PFGLS** to get an entry from the internal table generated by **PFLST**. Call **PFDLS** to release the table. **PFLST** automatically releases tables created by previous calls.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFCD**, **PFDLS**, **PFGLS**, **PFLN**, **PFMKDR**, **PFPWD**.

```
integer PFLST(integer fileid,
              integer npath,
              char *path,
              integer ntype,
              char *type,
              integer num)
```

Example

```
      integer pflst, pfdls
      integer fileid, num
             .
             .
             .
c generate a table of all directories in the current directory
      if (pflst(fileid, 0, '', 9, 'Directory', num) .eq. 0)
     $     call errproc
             .
             .
             .
c generate a table of the variables of type char * in directory animals
      if (pflst(fileid, 7, 'animals', 6, 'char *', num) .eq. 0)
     $     call errproc
             .
             .
             .
```

6.21 PFMKDR — (PD_mkdir)

```
integer PFMKDR(integer fileid,
               integer nchr,
               char *dirname)
```

Create a new directory in a PDB file.

PDBLib supports an optional hierarchical directory structure inside PDB files. A directory or a variable in a directory may be specified by either a relative path or an absolute path. Slashes separate nodes in a path name. Absolute paths begin with a slash. Nodes consisting of two periods, "..", refer to the next higher level directory.

The arguments to **PFMKDR** are: *fileid*, an integer file identifier; *nchr*, an integer number of characters in string *dirname*; and *dirname*, an ASCII string containing the path name of the new directory.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFCD**, **PFDLS**, **PFGLS**, **PFLN**, **PFLST**, **PFPWD**.

```
integer PFMKDR(integer fileid,
               integer nchr,
               char *dirname)
```

Example

```
      integer pfmkdr
      integer fileid
             .
             .
             .
      if (pfmkdr(fileid, 16, '/animals/mammals') .eq. 0)
     $     call errproc
             .
             .
             .
```

6.22 PFOPEN — (PD_open)

```
integer PFOPEN(integer nchr,
               character name,
               character mode)
```

Open an existing PDB file or create a new PDB file. Depending on the value of *mode*, PDBLib attempts to open the file *name* in read-only binary mode, open the file in append binary mode, or create a new file in read-write binary mode.

Any string which begins with ‘r’ causes the file to be opened in read-only mode, any string beginning with ‘a’ causes the file to be opened in append mode, and any string beginning with ‘w’ causes a file to be created in read-write mode.

Next, the beginning of the file is searched for the header which identifies the file as having been generated by PDBLib. The addresses of the structure chart and symbol table are then sought.

The structure chart from the file is read in. The structure chart contains information about data types (e.g. floats), their sizes in bytes and their structures, if any. By default there are six primitive data types that PDBLib knows about: short integers, integers, long integers, floating point numbers, double precision floating point numbers, characters, and pointers. The sizes of these types varies from machine to machine, but PDBLib hides this from the user.

The symbol table from the file is read in. The symbol table contains the list of variables in the file, their types as defined in the structure chart, and dimensioning information for arrays. Each read from the file first consults the symbol table to see if the requested variable is present in the PDB file.

Both the structure chart and the symbol table are implemented as hash tables, although their shapes are different. This makes lookups as efficient as possible given an unknown amount of data in the file.

The arguments to **PFOPEN** are: *nchr*, the number of characters in the string containing the file name; *name*, an ASCII string, which is the name of the file to be opened/created; and *mode*, an ASCII string, which is the mode (either ‘w’ for create, ‘r’ for read-only or ‘a’ for append).

The function returns an integer identifier for the PDB file opened/created. This identifies the particular file to PDBLib. As such, if it is overwritten the file is lost. The number of PDB files which can be open simultaneously is machine or operating system dependent, but each open PDB file has a unique identifier associated with it.

If any aspect of the PDB file opening process fails 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

```
integer PFOPEN(integer nchr,
               character name,
               character mode)
```

Example

```
      integer pfopen
      integer fileid
             .
             .
             .
      fileid = pfopen(8, 'file.pdb', 'r')
      if (fileid .eq. 0)
     $     call errproc
             .
             .
             .
```

6.23 PFPTRD — (PD_read_alt)

```
integer PFPTRD(integer fileid,
               integer nchr,
               character name,
               void space
               integer ind)
```

Read all or part of a data entry from an open PDB file. The symbol table of the given PDB file is searched for the given name and if it is found the information there is used to read the proper number of bytes from the file, do any conversions, and put the result in memory pointed to by *var*.

The elements to be read from an array are designated by start, stop, step triples (one for each dimension) passed in the argument *ind*.

The arguments to **PFPTRD** are: *fileid*, an integer identifier which designates the PDB file from which to attempt the read; *nchr*, an integer number of characters in the *name* string; *name*, an ASCII string containing the name of the variable to lookup; *space*, the location where the data is to be placed; and *ind*, an array of integers which specify the elements to be read.

Note: In each **PFPTRD** operation, the type of *space* must be a pointer to the type of *name*.

Note: When reading part of a variable, especially a structured variable, the path to the desired part must contain one array reference for each level of indirection traversed.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFREAD**, **PFRDAS**, and **PFRDAD**.

```
integer PFPTRD(integer fileid,
               integer nchr,
               character name,
               void space
               integer ind)
```

Example

```
      integer pfptrd
      integer fileid, ind(3)
      real xodd(10)
            .
            .
            .
c     read the first 10 odd elements of x into array xodd
      ind(1) = 1
      ind(2) = 20
      ind(3) = 2
      if (pfptrd(fileid, 1, 'x', xodd, ind) .eq. 0)
     $     call errproc
            .
            .
            .
```

6.24 PFPWD

```
integer PFPWD(integer fileid,
              integer nchr,
              char *dirname)
```

Get the current directory for a PDB file.

PDBLib supports an optional hierarchical directory structure inside PDB files. A directory or a variable in a directory may be specified by either a relative path or an absolute path. Slashes separate nodes in a path name. Absolute paths begin with a slash. Nodes consisting of two periods, "..", refer to the next higher level directory.

The arguments to **PFPWD** are: *fileid*, an integer file identifier; *nchr*, an integer to contain the number of characters returned in string *dirname*; and *dirname*, an ASCII string to contain the path name of the current directory.

This function copies the path name of a directory into the *dirname*, if successful. The application must ensure that *dirname* is large enough to contain the longest directory name (although PDBLib permits names of arbitrary length, 256 characters would probably be more than enough).

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFCD**, **PFDLS**, **PFGLS**, **PFLN**, **PFLST**, **PFMKDR**.

```
integer PFPWD(integer fileid,
              integer nchr,
              char *dirname)
```

Example

```
      integer pfpwd
      integer fileid, nchr
      character dirname(256)
             .
             .
             .
      if (pfpwd(fileid, nchr, dirname) .eq. 0)
     $     call errproc
             .
             .
             .
```

6.25 PFRATT — (PD_rem_attribute)

```
integer PFRATT(integer fileid,
               integer na
               char *attr)
```

Remove the specified attribute. PDBLib will remove the values of this attribute which may be associated with entities in the file.

The model of an attribute in PDBLib is an entity that has a name and type. The two supported operations on attributes are to create them and to remove them. An entity in a PDB file can be assigned an attribute value simply by calling **PFSVAT**. The only association between an entry in a PDB file and any attribute is made by the name in the attribute table and the entry in the symbol table. In particular, this mechanism allows the application developer to define and use entities in a PDB file solely in terms of attributes.

The arguments to **PFRATT** are: *fileid*, an integer identifier which designates the PDB file from which to remove the attribute; *na*, an integer number of characters in the *attr* string; and *attr*, an ASCII string containing the name of the attribute.

The return value is 1 if successful and 0 otherwise.

See also **PFDATT**, **PFGVAT**, and **PFSVAT**.

```
integer PFRATT(integer fileid,
               integer na
               char *attr)
```

Example

```
      integer pfratt
      integer fileid
             .
             .
             .
      if (pfratt(fileid, 4, ‘date’) .eq. 0)
     $     call errproc
             .
             .
             .
```

6.26 PFRDAD — (PD_read_as_alt)

```
integer PFRDAD(integer fileid,
               integer nchr,
               character name,
               integer ntype,
               character intype,
               void space
               integer ind)
```

Read all or part of a data entry from an open PDB file as data type *intype*. The symbol table of the given PDB file is searched for the given name and if it is found the information there is used to read the proper number of bytes from the file, do any conversions, and put the result in memory pointed to by *space*.

The elements to be read from an array are designated by start, stop, step triples (one for each dimension) passed in the argument *ind*.

The arguments to **PFRDAD** are: *fileid*, an integer identifier which designates the PDB file from which to attempt the read; *nchr*, an integer number of characters in the *name* string; *name*, an ASCII string containing the name of the variable to lookup; *ntype*, an integer number of characters in the *intype* string; *intype*, an ASCII string specifying the type of data to which *space* points; *space*, the location where the data is to be placed; and *ind*, an array of integers which specify the elements to be read.

Note: In each **PFRDAD** operation, the type of *space* must be a pointer to the type of *intype*.

Note: When reading part of a variable, especially a structured variable, the path to the desired part must contain one array reference for each level of indirection traversed.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFPTRD**, **PFRDAS**, and **PFREAD**.

```
integer PFRDAD(integer fileid,
               integer nchr,
               character name,
               integer ntype,
               character intype,
               void space
               integer ind)
```

Example

```
      integer pfrdad
      integer fileid, ind(3)
      real*8 xdd(10)
            .
            .
            .
c     read the first 10 elements of float x into array xdd
c     as type double.

      ind(1) = 1
      ind(2) = 10
      ind(3) = 1
      if (pfrdad(fileid, 1, 'x', 6, `double', xdd, ind) .eq. 0)
     $     call errproc
            .
            .
            .
```

6.27 PFRDAS — (PD_read_as)

```
integer PFRDAS(integer fileid,
               integer nchr,
               character name,
               integer ntype,
               character intype,
               void space)
```

Read data from an open PDB file as data type *intype*. The symbol table of the given PDB file is searched for the given name and if it is found the information there is used to read the proper number of bytes from the file, do any conversions, and put the result in memory pointed to by *space*.

The arguments to **PFRDAS** are: *fileid*, an integer identifier which designates the PDB file from which to attempt the read; *nchr*, an integer number of characters in the *name* string; *name*, an ASCII string containing the name of the variable to lookup; *ntype*, an integer number of characters in the *intype* string; *intype*, an ASCII string specifying the type of data to which *space* points; and *space*, the location where the data is to be placed.

Note: In each **PFRDAS** operation, the type of *space* must be a pointer to the type of *intype*.

Note: When reading part of a variable, especially a structured variable, the path to the desired part must contain one array reference for each level of indirection traversed.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFPTRD**, **PFRDAD**, and **PFREAD**.

```
integer PFRDAS(integer fileid,
               integer nchr,
               character name,
               integer ntype,
               character intype,
               void space)
```

Example

```
      integer pfrdas
      integer fileid
      real*8 xx(20)
            .
            .
            .
c     read array x into array xx as type double
      if (pfrdas(fileid, 1, 'x', 6, 'double', xx) .eq. 0)
     $     call errproc
            .
            .
            .
```

6.28 PFREAD — (PD_read)

```
integer PFREAD(integer fileid,
               integer nchr,
               character name,
               void space)
```

Read data from an open PDB file. The symbol table of the given PDB file is searched for the given name and if it is found the information there is used to read the proper number of bytes from the file, do any conversions, and put the result in memory pointed to by *space*.

The arguments to **PFREAD** are: *fileid*, an integer identifier which designates the PDB file from which to attempt the read; *nchr*, an integer number of characters in the *name* string; *name*, an ASCII string containing the name of the variable to lookup; and *space*, the location where the data is to be placed.

Note: In each **PFREAD** operation, the type of *space* must be a pointer to the type of *name*.

Note: When reading part of a variable, especially a structured variable, the path to the desired part must contain one array reference for each level of indirection traversed.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFPTRD**, **PFRDAD**, and **PFRDAS**.

```
integer PFREAD(integer fileid,
               integer nchr,
               character name,
               void space)
```

Example

```
      integer pfread
      integer fileid
      real x(20)
      common /abc/ a(2), b, c(2,2:4)
      real a, b, c
      common /jkl/ j, k, l
            .
            .
            .
c     read array x
      if (pfread(fileid, 1, 'x', x) .eq. 0)
     $     call errproc

c     read first element of member c of structure abc
      if (pfread(fileid, 10, 'abc.c(1,2)', c) .eq. 0)
     $     call errproc

c     read entire structure jkl
      if (pfread(fileid, 3, 'jkl', j) .eq. 0)
     $     call errproc
            .
            .
            .
```

6.29 PFSVAT — (PD_set_attribute)

```
integer PFSVAT(integer fileid,
               integer nv,
               char *var,
               integer na
               char *attr,
               void value)
```

Set the value of the specified attribute for the named entity.

The model of an attribute in PDBLib is an entity that has a name and type. The two supported operations on attributes are to create them and to remove them. An entity in a PDB file can be assigned an attribute value simply by calling **PFSVAT**. The only association between an entry in a PDB file and any attribute is made by the name in the attribute table and the entry in the symbol table. In particular, this mechanism allows the application developer to define and use entities in a PDB file solely in terms of attributes.

The arguments to **PFSVAT** are: *fileid*, an integer identifier which designates the PDB file in which the attribute is being set; *nv*, an integer number of characters in the *var* string; *var*, an ASCII string containing the name of an entity; *na*, an integer number of characters in the *attr* string; *attr*, an ASCII string containing the name of the attribute being set; and *value* containing the attribute value.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFDATT**, **PFRATT**, and **PFGVAT**.

```
integer PFSVAT(integer fileid,
               integer nv,
               char *var,
               integer na
               char *attr,
               void value)
```

Example

```
      integer pfsvat
      integer scftcs
      integer fileid
      character*80 dt
            .
            .
            .
      scftcs(dt, ‘Mon March 27, 1921’, 18)
      if (pfsvat(fileid, 3, ‘foo’, 4, ‘date’, dt) .eq. 0)
     $     call errproc
            .
            .
            .
```

6.30 PFTRGT — (PD_target)

```
integer PFTRGT(integer is,
               integer ia)
```

Write the next PDB file to be created according to the specified data standard and alignment. PDBLib has a general binary data conversion mechanism called parametrized data conversion (PDC). An integer type is described by one set of parameters and a floating point type is described by another. A general purpose conversion routine takes the description of the input type and a description of the desired output type and does the conversion. In this way, PDBLib avoids an N^2 increase in data conversion routines as it ports to new machines. In fact, the number of data standards and alignments grows more slowly than N because many machines share common formats.

An additional advantage to PDC is that by specifying a format involving the minimal number of bits to represent the data for a file, PDBLib can carry out a large class of data compressions.

PDBLib carries several predefined **data_standard** and **data_alignment** structures which describe a large number of binary data formats. In the C API these may be directly referenced and their names are given here for completeness.

Data standards: **IEEEA_STD**, **IEEEB_STD**, **IEEEC_STD**, **INTELA_STD**, **INTELB_STD**, **VAX_STD**, and **CRAY_STD**.

Data alignments: **DEF_ALIGNMENT**, **SPARC_ALIGNMENT**, **MIPS_ALIGNMENT**, **RS6000_ALIGNMENT**, **68000_ALIGNMENT**, **INTEL_ALIGNMENT**, **CRAY_ALIGNMENT**, and **UNICOS_ALIGNMENT**.

In the FORTRAN API, these structures are placed in two arrays and the indices into these arrays are passed into **PFTRGT** to specify the binary format to be targeted.

The list of standards is:

1 - IEEEA_STD

2 - IEEEB_STD

3 - IEEEC_STD

4 - INTELA_STD

5 - INTELB_STD

6 - VAX_STD

7 - CRAY_STD

The list of alignments is:

1 - M68000_ALIGNMENT
2 - SPARC_ALIGNMENT
3 - MIPS_ALIGNMENT
4 - INTEL_ALIGNMENT
5 - DEF_ALIGNMENT
6 - CRAY_ALIGNMENT
7 - UNICOS_ALIGNMENT
8 - RS6000_ALIGNMENT

Some common configurations are:

Motorola	- PFTRGT(1, 1)
SPARC	- PFTRGT(1, 2)
MIPS, SGI	- PFTRGT(1, 3)
IBM RS6000	- PFTRGT(1, 8)
Mac/Think C	- PFTRGT(2, 1)
Mac/MPW	- PFTRGT(3, 1)
DOS	- PFTRGT(4, 4)
Intel 80x86 UNIX	- PFTRGT(5, 4)
DEC Vax	- PFTRGT(6, 5)
DEC 3100	- PFTRGT(5, 3)
NLTSS Cray	- PFTRGT(7, 6)
UNICOS Cray	- PFTRGT(7, 7)

The argument, *is*, is an index corresponding one of the **data_standard** structures listed above, and the argument, *ia*, is an index corresponding to one of the **data_alignment** structures above. See the section on Data Structures later on in this manual.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

```
integer PFTRGT(integer is,
               integer ia)
```

Example

```
      integer pftrgt
      integer is, ia
             .
             .
             .
c     set target architecture (MIPS, SGI)
c     IEEEA_STD
      is = 1
c     MIPS_ALIGNMENT
      ia = 3
      if (pftrgt(is, ia) .eq. 0)
     $     call errproc
             .
             .
             .
```

6.31 PFVART

```
integer PFVART(integer fileid,
               integer order,
               integer nvars)
```

Generate an internal table of variables in the specified PDB file. With subsequent calls to **PFGVAR**, each entry can be obtained one at a time by ordinal index. The table is sorted according to a specified scheme. The current choices are alphabetic and by disk address.

The arguments to this function are: *fileid,* an integer which identifies the **PDBfile**; *order*, an integer specifying the sort ordering; and *nvars*, an integer in which the number of variables in the file is returned. The choices for *order* are: 1, for an alphabetic sort; and 2, for a disk address order sort.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFDVAR** and **PFGVAR**.

```
integer PFVART(integer fileid,
               integer order,
               integer nvars)
```

Example

```
      integer pfvart
      integer fileid, nvar, order
             .
             .
             .
c     create table of variable names in alphabetic order
      order = 1
      if (pfvart(fileid, order, nvar) .eq. 0)
     $     call errproc
             .
             .
             .
```

6.32 PFWIMA

```
integer PFWIMA(integer fileid,
               integer nchr,
               character name,
               integer pkn,
               integer pkx,
               integer pln,
               integer plx,
               real*8 data,
               real*8 pxn,
               real*8 pxx,
               real*8 pyn,
               real*8 pyx,
               integer pim)
```

Build a **PD_image** structure out of the given input data and write it to a PDB file.

Rationale: The **PD_image** structure is a useful and general purpose representation of a raster image. The nature of the generalization is that the values in the raster are floating point numbers. So in addition to the standard data sets that can be rasterized, the **PD_image** can be used to display the computational matrix of some system of equations, for example. This function is a convenient way for FORTRAN programs to put out their data into PDB files as **PD_image**'s for later visualization and processing by other programs. It allows a rectangular subset of a two dimensional array to be specified for the **PD_image**.

The calling application must keep track of how many **PM_image**'s have been written to each file. PDBLib will write each **PM_image** under the name composed of the string, 'Image', and the integer *pim*. For example if *pim* is 9, the **PM_image** will be written under the name 'Image9'. If the application passes the same value for *pim* more than once only the last one will survive in the symbol table even though the data for each **PM_image** will persist in the file!

The arguments to **PFWIMA** are: *fileid*, an integer identifier which designates the PDB file to which to attempt to write; *nchr*, an integer number of characters in *name*; *name*, an ASCII string containing the name of the image; *pkn*, the integer minimum column index of the data array; *pkx*, the integer maximum column index of the data array; *pln*, the integer minimum row index of the data array; *plx*, the integer maximum row index of the data array; *data*, an array of real*8 values containing the image data; *pxn*, a real*8 value specifying the minimum column index in image; *pxx*, a real*8 value specifying the maximum column index in image; *pyn*, a real*8 value specifying the minimum row index in image; *pyx*, a real*8 value specifying the maximum row index in image; and *pim*, a counter specifying the number of the image being written out.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

```
integer PFWIMA(integer fileid,
               integer nchr,
               character name,
               integer pkn,
               integer pkx,
               integer pln,
               integer plx,
               real*8 data,
               real*8 pxn,
               real*8 pxx,
               real*8 pyn,
               real*8 pyx,
               integer pim)
```

Example

```
      integer pfwima
      integer fileid, k, l
c     these arguments to pfwima must have 8 byte element size
      double precision xmin, xmax, ymin, ymax, data(0:10,0:10)
             .
             .
             .
      xmin = 0.
      xmax = 10.
      ymin = 0.
      ymax = 10.
      do 100 l = 0, 10
         do 101 k = 0, 10
            data(k, l) = (k - 5)**2 + (l - 5)**2
 101     continue
 100  continue

      if (pfwima(fileid, 10, 'Test image', 0, 10, 0, 10,
     $     data, xmin, xmax, ymin, ymax, 1) .eq. 0)
     $     call errproc
             .
             .
             .
```

6.33 PFWMAP

```
integer PFWMAP(integer fileid,
               character dname,
               integer dp,
               real*8 dm,
               character rname,
               integer rp,
               real*8 rm,
               integer pim)
```

Build a **PM_mapping** structure out of the given input data and write it to a PDB file.

Rationale: The **PM_mapping** structure is a convenient medium of exchange between data production systems such as simulation codes, storage systems such as PDBLib, and visualization systems such as PDBView. This function is a convenient way for FORTRAN programs to put out their data into PDB files as **PM_mapping**'s for later visualization and processing by other programs.

The **PM_mapping** is a structure with two main parts: a domain and a range. These two parts are in turn represented by a structure called a **PM_set**. Because they are both represented as the same type of data object, they are specified similarly in **PFWMAP**. For each of the domain and range sets the following information is given: a name; an array of integer quantities specifying such information as the dimensionality of the set, the dimensionality of the elements, the number of elements, and so on; and a linear array containing the elements of the set.

The entries in the arrays *dp* and *rp* are as follows:

1	- the number of characters in the corresponding set name
2	- the dimensionality of the set, *nd*
3	- the dimensionality of the set elements, *nde*
4	- the number of elements in the set, *ne*
5 thru 5+ *nd*-1	- the sizes in each dimension

The layout of the set elements in *dm* and *rm* is:

1 thru *ne*	- values of the first component
ne+1 thru 2**ne*	- values of the second component
...	- values of components
(*nde*-1)**ne*+1 thru *nde***ne*	- values of the *nde*'th component

The calling application must keep track of how many **PM_mapping**'s have been written to each file. PDBLib will write each **PM_mapping** under the name composed of the string, 'Mapping', and the integer *pim*. For example if *pim* is 9, the **PM_mapping** will be written under the name ‘Mapping9’. If the application passes the same value for *pim* more than once only the last one will survive in the symbol table even though the data for each **PM_mapping** will persist in the file!

The arguments to **PFWMAP** are: *fileid*, an integer identifier which designates the PDB file to which to attempt to write; *dname*, an ASCII string containing the name of the domain set; *dp*, an integer array of parameters defining the domain set; *dm*, an array of real*8 values containing the set elements component by component; *rname*, an ASCII string containing the name of the range set; *rp*, an integer array of parameters defining the range set; *rm*, an array of real*8 values containing the range elements component by component; and *pim* a counter specifying the number of the mapping being written out.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

```
integer PFWMAP(integer fileid,
               character dname,
               integer dp,
               real*8 dm,
               character rname,
               integer rp,
               real*8 rm,
               integer pim)
```

Example

```
      integer pfwmap
      integer fileid, dp(5), rp(5)
      double precision dm(0:99), rm(0:99)
             .
             .
             .
      dp(1) = 6
      dp(2) = 1
      dp(3) = 1
      dp(4) = 100
      dp(5) = 100
      rp(1) = 6
      rp(2) = 1
      rp(3) = 1
      rp(4) = 100
      rp(5) = 100

      do 100 i = 0, 99
         dm(i) = 6.28*float(i)/99.
         rm(i) = sin(6.28*float(i)/99.)
 100  continue

      if (pfwmap(fileid, 'Domain', dp, dm, 'Range', rp, rm, 0)
     $    .eq. 0)
     $     call errproc
             .
             .
             .
```

6.34 PFWRAD — (PD_write_as_alt)

```
integer PFWRAD(integer fileid,
               integer nchr,
               character name,
               integer ntypin,
               character intype,
               integer ntypout,
               character outtype,
               void space,
               integer nd,
               integer ind)
```

Write the data of type *intype* to the PDB file as data of type *outtype*.

The rationale for this function is:
1) that in some situations, it is desirable to not only convert the formats of data of a specified type, but to convert between types. An example that occurs in practice often enough is converting a 32 bit int to a 32 bit long on a machine which only has a 16 bit int.
2) that in some situations, it is desirable to be able to specify the dimensions without building them into an ASCII string.

The arguments to **PFWRAD** are: *fileid*, an integer identifier which designates the PDB file to which to write; *nchr*, an integer number of characters in the *name* string; *name*, an ASCII string containing the name of the variable to install in the symbol table; *ntypin*, an integer number of characters in the *intype* string; *intype*, an ASCII string specifying the variable type in *space*; *ntypout*, an integer number of characters in the *outtype* string; *outtype*, an ASCII string specifying the variable type in the PDB file; *space*, the data to be written; *nd*, an integer number of dimensions; and *ind*, an array of *nd* integer triples containing the minimum and maximum index and the stride for each dimension.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFAPAS**, **PFAPAD**, **PFAPPA**, **PFAPPD**, **PFWRAS**, **PFWRTA**, and **PFWRTD**.

```
integer PFWRAD(integer fileid,
               integer nchr,
               character name,
               integer ntypin,
               character intype,
               integer ntypout,
               character outtype,
               void space,
               integer nd,
               integer ind)
```

Example

```
      integer pfwrad
      integer fileid, nd, ind(6)
      real*8 c(2,2:4)
            .
            .
            .
c     write array c of type double as type float
      nd = 2
      ind(1) = 1
      ind(2) = 2
      ind(3) = 1
      ind(4) = 2
      ind(5) = 4
      ind(6) = 1
      if (pfwrad(fileid, 1, 'c', 6, 'double',
     $                          5, 'float', c, nd, ind) .eq. 0)
     $     call errproc
            .
            .
            .
```

6.35 PFWRAN

```
integer PFWRAN(integer fileid,
               character dname,
               integer nchr,
               character rname,
               integer rp,
               real*8 rm,
               integer pim)
```

Build a **PM_mapping** structure out of the given input data and write it to a PDB file.

Rationale: The **PM_mapping** structure is a convenient medium of exchange between data production systems such as simulation codes, storage systems such as PDBLib, and visualization systems such as PDBView. This function is a convenient way for FORTRAN programs to put out their data into PDB files as **PM_mapping**'s for later visualization and processing by other programs.

The **PM_mapping** is a structure with two main parts: a domain and a range. These two parts are in turn represented by a structure called a **PM_set**. In many cases a number of **PM_mapping**'s share a common domain set. It is therefore more efficient to write the unique domain sets out separately and use **PFWRAN** to write out the **PM_mapping**'s without their domains. Post processor codes such as PDBView (by definition) know how to put the full **PM_mapping** back together. **Note:** the domain name given for **PFWRAN** must be the same as the domain name passed to the corresponding **PFWSET** call.

For each range set the following information is given: a name; an array of integer quantities specifying such information as the dimensionality of the set, the dimensionality of the elements, the number of elements, and so on; and a linear array containing the elements of the set.

The entries in the array *rp* are as follows:

1	- the number of characters in the corresponding set name
2	- the dimensionality of the set, *nd*
3	- the dimensionality of the set elements, *nde*
4	- the number of elements in the set, *ne*
5 thru 5+ *nd*-1	- the sizes in each dimension

The layout of the set elements in *rm* is:

1 thru *ne*	- values of the first component
ne+1 thru 2***ne*	- values of the second component
...	- values of components
(*nde*-1)***ne*+1 thru *nde***ne*	- values of the *nde*'th component

The calling application must keep track of how many **PM_mapping**'s have been written to each file. PDBLib will write each **PM_mapping** under the name composed of the string, 'Mapping', and the integer *pim*. For example if *pim* is 9, the **PM_mapping** will be written under the name 'Mapping9'. If the application passes the same value for *pim* more than once only the last one will survive in the symbol table even though the data for each **PM_mapping** will persist in the file.

The arguments to **PFWRAN** are: *fileid*, an integer identifier which designates the PDB file to which to attempt to write; *dname*, an ASCII string containing the name of the domain set; *nchr*, an integer number of characters in *dname*; *rname*, an ASCII string containing the name of the range set; *rp*, an integer array of parameters defining the range set; *rm*, an array of real*8 values containing the range elements component by component; and *pim* a counter specifying the number of the mapping being written out.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

```
integer PFWRAN(integer fileid,
               character dname,
               integer nchr,
               character rname,
               integer rp,
               real*8 rm,
               integer pim)
```

Example

```
      integer pfwran
      integer fileid, i, pim, rp(5)
      double precision rm(0:99)
             .
             .
             .
      pim = 0
      rp(1) = 6
      rp(2) = 1
      rp(3) = 1
      rp(4) = 100
      rp(5) = 100

      do 100 i = 0, 99
         rm(i) = sin(6.28*float(i)/99.)
 100  continue

c     ‘Domain’ written previously by pfwset
      if (pfwran(fileid, ‘Domain’, 6, ‘Range’, rp, rm, pim)
     $    .eq. 0)
     $     call errproc
             .
             .
             .
```

6.36 PFWRAS — (PD_write_as)

```
integer PFWRAS(integer fileid,
               integer nchr,
               character name,
               integer ntypin,
               character intype,
               integer ntypout,
               character outtype,
               void space)
```

Write the data of type *intype* to the PDB file as data of type *outtype*.

The rationale for this function is that in some situations, it is desirable to not only convert the formats of data of a specified type, but to convert between types. An example that occurs in practice often enough is converting a 32 bit int to a 32 bit long on a machine which only has a 16 bit int.

The arguments to **PFWRAS** are: *fileid*, an integer identifier which designates the PDB file to which to write; *nchr*, an integer number of characters in the *name* string; *name*, an ASCII string containing the name of the variable to install in the symbol table; *ntypin*, an integer number of characters in the *intype* string; *intype*, an ASCII string specifying the variable type in *space*; *ntypout*, an integer number of characters in the *outtype* string; *outtype*, an ASCII string specifying the variable type in the PDB file; and *space*, the data to be written.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**

See also **PFAPAS**, **PFAPAD**, **PFAPPA**, **PFAPPD**, **PFWRAD**, **PFWRTA**, and **PFWRTD**.

```
integer PFWRAS(integer fileid,
               integer nchr,
               character name,
               integer ntypin,
               character intype,
               integer ntypout,
               character outtype,
               void space)
```

Example

```
      integer pfwras
      integer fileid
      real*8 x(20)
                    .
             .
             .
c     write array x of type double as type float
      if (pfwras(fileid, 5, ‘x(20)’, 6, ‘double’,
     $                              5, ‘float’, x) .eq. 0)
     $     call errproc
             .
             .
             .
```

6.37 PFWRTA — (PD_write)

```
integer PFWRTA(integer fileid,
               integer nchr,
               character name,
               integer ntype,
               character type,
               void space)
```

Write data to a PDB file. Before writing data to the PDB file an entry is prepared for the symbol table consisting of the name, the type, the dimension information, the disk address to which the data will be written, and the total number of bytes as computed with the help of the structure chart. After the entry is installed in the symbol table the data from memory is converted (only if the target machine type is different from the current machine type) and then written out to disk starting at the current disk address.

The primitive data types which the PDBLib system knows about by default are: 'short', 'integer', 'long', 'float', 'double', and 'char' for short integer, integer, long integer, floating point or real*8 number, double precision floating point number, and character or single byte respectively.

Rationale: When writing out scalar variables (i.e. non-dimensioned variables - structured variables are scalars unless specifically dimensioned) or arrays with fixed dimensions this function is the most convenient to use. Dimension information can be encoded in the name of the entry. PDBLib will strip that information off the name before creating the symbol table entry.

Dimensions can be given in two ways. If the variable was not previously defined and a single number is specified for a given dimension, it is interpreted as the length of that dimension and the default offset value for the PDB file is taken as the minimum value. In this case the maximum value is the default offset plus the minimum value minus one. (If the variable was previously defined, a single number is interpreted as a reference to that one dimension index value.) Alternatively, the minimum value, maximum value, and stride (separated by colons) may be specified. The stride is optional and defaults to 1. For example:

pfwrta(pid, 10, 'u(30,1:10)', 5, 'float', u)

The arguments to **PFWRTA** are: *fileid*, an integer identifier which designates the PDB file to which to write; *nchr*, an integer number of characters in the *name* string; *name*, an ASCII string containing the name of the variable to install in the symbol table; *ntype*, an integer number of characters in the *type* string; *type*, an ASCII string specifying the variable *t*ype; and *space*, the data to be written.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFAPAS**, **PFAPAD**, **PFAPPA**, **PFAPPD**, **PFWRAD**, **PFWRAS**, and **PFWRTD**.

```
integer PFWRTA(integer fileid,
               integer nchr,
               character name,
               integer ntype,
               character type,
               void space)
```

Example

```
      integer pfwrta
      integer fileid
      real x(20)
      common /abc/ a(2), b, c(2,2:4)
      real a, b, c
            .
            .
            .
c     write array x
      if (pfwrta(fileid, 5, 'x(20)', 5, 'float', x) .eq. 0)
     $     call errproc

c     write entire structure abc (previous defined with pfdefs)
      if (pfwrta(fileid, 3, 'abc', 3, 'abc', a) .eq. 0)
     $     call errproc
            .
            .
            .
```

6.38 PFWRTD — (PD_write_alternate)

```
integer PFWRTD(integer fileid,
               integer nchr,
               character name,
               integer ntype,
               character type,
               void space,
               integer nd,
               integer ind)
```

Write data to a PDB file. Before writing data to the PDB file an entry is prepared for the symbol table consisting of the name, the type, the dimension information, the disk address to which the data will be written, and the total number of bytes as computed with the help of the structure chart. After the entry is installed in the symbol table the data from memory is converted (only if the target machine type is different from the current machine type) and then written out to disk starting at the current disk address.

The primitive data types which the PDBLib system knows about by default are: ‘short’, ‘integer’, ‘long’, ‘float’, ‘double’, and ‘char’ for short integer, integer, long integer, floating point or real number, double precision floating point number, and character or single byte respectively.

The rationale for this function is that in some situations, it is desirable to be able to specify the dimensions without building them into an ASCII string.

The arguments to **PFWRTD** are: *fileid*, an integer identifier which designates the PDB file to which to write; *nchr*, an integer number of characters in the *name* string; *name*, an ASCII string containing the name of the variable to install in the symbol table; *ntype*, an integer number of characters in the *type* string; *type*, an ASCII string specifying the variable type; *space*, the data to be written; *nd*, an integer number of dimensions; and *ind*, an array of *nd* integer triples containing the minimum and maximum index and the stride for each dimension.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

See also **PFAPAS**, **PFAPAD**, **PFAPPA**, **PFAPPD**, **PFWRAD**, **PFWRAS**, and **PFWRTA**.

```
integer PFWRTD(integer fileid,
               integer nchr,
               character name,
               integer ntype,
               character type,
               void space,
               integer nd,
               integer ind)
```

Example

```
      integer pfwrtd
      integer fileid, nd, ind(6)
      real c(2,2:4)
            .
            .
            .
c     write array c
      nd = 2
      ind(1) = 1
      ind(2) = 2
      ind(3) = 1
      ind(4) = 2
      ind(5) = 4
      ind(6) = 1
      if (pfwrtd(fileid, 1, 'c', 5, 'float', c, nd, ind) .eq. 0)
     $     call errproc
            .
            .
            .
```

6.39 PFWSET)

```
integer PFWSET(integer fileid,
               character dname,
               integer dp,
               real*8 dm)
```

Build a **PM_set** structure out of the given input data and write it to a PDB file.

Rationale: The **PM_set** structure is a fundamental component of the **PM_mapping** structure which is a convenient medium of exchange between data production systems such as simulation codes, storage systems such as PDBLib, and visualization systems such as PDBView. Although the function **PFWMAP** most conveniently writes a mapping out to a PDB file, it does not make the best use of limited storage space. In many cases a number of **PM_mapping**'s share a common domain set. It is therefore more efficient to write the unique domain sets out separately and use **PFWRAN** to write out the **PM_mapping**'s without their domains. Post processor codes such as PDBView (by definition) know how to put the full **PM_mapping** back together. **Note:** the domain name given for **PFWSET** must be the same as the domain name passed to the corresponding **PFWRAN** call!

For each **PM_set** the following information is given: a name; an array of integer quantities specifying such information as the dimensionality of the set, the dimensionality of the elements, the number of elements, etc.; and a linear array containing the elements of the set.

The entries in the array *dp* are as follows:

1	- the number of characters in the corresponding set name
2	- the dimensionality of the set, *nd*
3	- the dimensionality of the set elements, *nde*
4	- the number of elements in the set, *ne*
5 thru 5+*nd*-1	- the sizes in each dimension

The layout of the set elements in *dm* is:

1 thru *ne*	- values of the first component
ne+1 thru 2**ne*	- values of the second component
...	- values of components
(*nde*-1)**ne*+1 thru *nde***ne*	- values of the *nde*'th component

The arguments to **PFWSET** are: *fileid*, an integer identifier which designates the PDB file to which to write; *dname*, an ASCII string containing the name of the **PM_set**; *dp*, an integer array of parameters defining the **PM_set**; and *dm*, an array of real*8 values containing the set elements component by component.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function **PFGERR**.

```
integer PFWSET(integer fileid,
               character dname,
               integer dp,
               real*8 dm)
```

Example

```
      integer pfwset
      integer fileid, i, dp(5)
      double precision dm(0:99)
             .
             .
             .
      dp(1) = 6
      dp(2) = 1
      dp(3) = 1
      dp(4) = 100
      dp(5) = 100

      do 100 i = 0, 99
         dm(i) = 6.28*float(i)/99.
 100  continue

      if (pfwset(fileid, 'Domain', dp, dm) .eq. 0)
     $     call errproc
             .
             .
             .
```

7.0 PDBLib Design Philosophy

7.1 Overview

Perhaps the most fundamental element in the design of PDBLib is the concept of modularity through abstraction barriers. In essence, the functionality is broken down into modular units and abstraction barriers are used to preserve the integrity of the units.

An abstraction barrier is simply a technique or device which allows a section of code to be written assuming other functions or routines are defined and that their internal workings are irrelevant to their use. In this way a routine or module can be changed without any other part of the code which uses it being affected (so long as the definition of its function does not change). Abstraction barriers are most effectively created by a careful choice of the basic functional units and by the interfaces between them.

For example, if all variables in the code were global there would be little or no chance of having any abstraction barriers at all. Similarly, monolithic functions which are defined to 'solve the worlds problems' do not lend themselves to the more easy maintenance that abstraction barriers afford a program. For a good discussion of the principles and applications of these ideas see Abelson and Sussman's marvelous book, The Structure and Interpretation of Computer Programs.

The main functional units in the PDBLib system are: the hash package which controls the maintenance of the data in both the symbol table and structure chart; the conversion package which handles all data format conversions; the data reading routine which is defined to bring data in from the disk, perform any necessary conversion, and store it in the specified location(s) of memory; and the data writing routine which does the reverse of the reading routine.

These units are put together in such a way that they are used over and over again in the process of doing any of the high level functions. In this way the code size was kept relatively small. Again this is one of the rewards of modularity.

7.2 Data Conversion and Compression

PDBLib has a quasi-universal data translation capability. It is called Parametrized Data Conversion (PDC). A set of parameters which characterizes a large set of integer and floating point formats was developed. It describes the byte size and order of integer types. For floating point data it describes the bit location and width of the sign, exponent, and mantissa field as well as the byte size and order of the data. Using this information a single integer conversion routine and a single floating point conversion routine handle all of the data conversions in PDBLib. The advantage of this approach is that there is no increase in the size of the library for each port to a new environment. Furthermore, it will allow future releases to auto-configure themselves to the machines on which they run. Another benefit is that a data representation may be targeted without regard to its implementation. This provides a vehicle for developing data representations, evaluating them, or using them in a highly abstract manner. The sole drawback of this approach is that it makes assumptions about the representation of data in computers. While the assumptions are general and the result of incorporating data representations outside these assumptions is more work on the library itself, it is philosophically unsatisfying to make any assumptions about how things are to be done or data to be represented. The other drawback is that by being general purpose the conversion routines are slightly slower than specific ones. This is more than made up for in the saving in library size and ease of porting the library.

Alternative data conversion strategies are either hub and spoke (such as Sun's XDR) or specific format to format. The latter suffers from an N^2 growth in the number of conversion routines where N is the number of machine/architectures which the library supports. On the other hand, hub and spoke strategies necessitate a conversion on each read or write operation.

PDC prevents the conversion problem from being N^2. At worst, the PDC method should grow like N in the number of parameter sets required. In practice, it is even better than that. Most computer systems today are based on a handful of CPU's which are the most constraining factor in binary data formats. For the convenience of the users of PDBLib, several **data_standard***'s* and **data_alignment***'s* are predefined by the library itself.

A significant advantage to PDC is that a class of data compression algorithms is implicit in the method. By simply describing a format which describes data in the correct range (up to a possible overall offset for each type), PDBLib can do all of the work to store and retrieve the data in a compressed form.

7.2.1 DATA_STANDARD'S

See the section on the **data_standard** structure for a detailed discussion of its members. The following is a list of the **data_standard**'s which PDBLib provides by default. The purpose is twofold: to help users identify target formats; and to guide users who wish to create their own **data_standard**'s.

The members of the **data_standard** are indicated in the template:

```
STAND
        {size of pointer,
         size and order of short,
         size and order of int,
         size and order of long,
         size, format, and order of float,
         size, format, and order of double}
```

The various variables indicated are defined by PDBLib.

```
DEF_STD
        {4,
         2, NORMAL_ORDER,
         4, NORMAL_ORDER,
         4, NORMAL_ORDER,
         4, def_float, def_float_order,
         8, def_double, def_double_order}
```

```
IEEEA_STD
        {4,
         2, NORMAL_ORDER,
         4, NORMAL_ORDER,
         4, NORMAL_ORDER,
         4, ieee_float, ieee_float_order,
         8, ieeea_double, ieeea_double_order}
```

IEEEB_STD

{4,
2, NORMAL_ORDER,
2, NORMAL_ORDER,
4, NORMAL_ORDER,
4, ieee_float, ieee_float_order,
12, ieeeb_double, ieeeb_double_order}

IEEEC_STD

{4,
2, NORMAL_ORDER,
4, NORMAL_ORDER,
4, NORMAL_ORDER,
4, ieee_float, ieee_float_order,
12, ieeeb_double, ieeeb_double_order}

INTELA_STD

{4,
2, REVERSE_ORDER,
2, REVERSE_ORDER,
4, REVERSE_ORDER,
4, intel_float, intel_float_order,
8, intel_double, intel_double_order}

INTELB_STD

{4,
2, REVERSE_ORDER,
4, REVERSE_ORDER,
4, REVERSE_ORDER,
4, intel_float, intel_float_order,
8, intel_double, intel_double_order},

```
VAX_STD
        {4,
         2, REVERSE_ORDER,
         4, REVERSE_ORDER,
         4, REVERSE_ORDER,
         4, vax_float, vax_float_order,
         8, vax_double, vax_double_order}

CRAY_STD
        {8,
         8, NORMAL_ORDER,
         8, NORMAL_ORDER,
         8, NORMAL_ORDER,
         8, cray_float, cray_float_order,
         8, cray_float, cray_float_order}
```

7.2.1.1 Floating Point Format Descriptor

The description of a floating point number consists of an array of 8 long integer parameters. The parameters are:

format[0] = # of bits per number

format[1] = # of bits in exponent

format[2] = # of bits in mantissa

format[3] = start bit of sign

format[4] = start bit of exponent

format[5] = start bit of mantissa

format[6] = high order mantissa bit

format[7] = bias of exponent

The following floating point format descriptors are defined in PDBLib and used in the **data_standard**'s described in the last section:

```
ieee_float    = {32L, 8L, 23L, 0L, 1L, 9L, 0L, 0x7FL}
ieeea_double  = {64L, 11L, 52L, 0L, 1L, 12L, 0L, 0x3FFL}
ieeeb_double  = {96L, 15L, 64L, 0L, 1L, 32L, 1L, 0x3FFEL}
intel_float   = {32L, 8L, 23L, 0L, 1L, 9L, 0L, 0x7FL}
intel_double  = {64L, 11L, 52L, 0L, 1L, 12L, 0L, 0x3FFL}
cray_float    = {64L, 15L, 48L, 0L, 1L, 16L, 1L, 0x4000L}
```

Note: There are several variants of floating type format on the VAX. Accordingly the user must decide which one to use and PDBLib has the descriptions:

When using GFLOATs

```
vax_float     = {32L, 8L, 23L, 0L, 1L, 9L, 0L, 0x81L}
vax_double    = {64L, 11L, 52L, 0L, 1L, 12L, 0L, 0x401L}
```

otherwise

```
vax_float     = {32L, 8L, 23L, 0L, 1L, 9L, 0L, 0x81L}
vax_double    = {64L, 8L, 55L, 0L, 1L, 9L, 0L, 0x81L}
```

7.2.1.2 Byte Ordering

There is much discussion in the literature about little endian and big endian machines. Those two refer to two possible byte orderings for binary data. That is not the most general way to talk about byte ordering however. In fact the VAX format exemplifies the need for generality. PDBLib simply uses an array of integers which describe the order of the bytes in memory relative to CPU's such as the Motorola and SPARC families.

ieee_float_order	= {1, 2, 3, 4}
ieeea_double_order	= {1, 2, 3, 4, 5, 6, 7, 8}
ieeeb_double_order	= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
intel_float_order	= {4, 3, 2, 1}
intel_double_order	= {8, 7, 6, 5, 4, 3, 2, 1}
vax_float_order	= {2, 1, 4, 3}
vax_double_order	= {2, 1, 4, 3, 6, 5, 8, 7}
cray_float_order	= {1, 2, 3, 4, 5, 6, 7, 8}

7.2.2 DATA_ALIGNMENTS

The term alignment refers to the fact that many CPU's require certain data types to begin at memory locations whose addresses are even multiples of some integer number of bytes. So for example, to say that the alignment of a double is 8 means that a double must begin at an address which is a multiple of 8.

Compilers hide this concept from almost all applications. PDBLib is one that must know data alignments precisely. It employs a structure called a **data_alignment** to record the alignments of the default primitive data types which PDBLib supports. See the discussion of the **data_alignment** structure in the section on data structures.

The following is the list of **data_alignment**'s which PDBLib provides automatically (applications can add their own as needed):

```
SPARC_ALIGNMENT   = {1, 4, 2, 4, 4, 4, 8}
MIPS_ALIGNMENT    = {1, 4, 2, 4, 4, 4, 8}
RS6000_ALIGNMENT  = {1, 4, 2, 4, 4, 4, 4}
CRAY_ALIGNMENT    = {8, 8, 8, 8, 8, 8, 8}
UNICOS_ALIGNMENT  = {1, 8, 8, 8, 8, 8, 8}
M68000_ALIGNMENT  = {1, 2, 2, 2, 2, 2, 2}
INTELA_ALIGNMENT  = {1, 2, 2, 2, 2, 2, 2}
INTELB_ALIGNMENT  = {4, 4, 4, 4, 4, 4, 4}
INTELC_ALIGNMENT  = {2, 4, 2, 4, 4, 4, 4}
DEF_ALIGNMENT     = {1, 4, 4, 4, 4, 4, 4}
```

8.0 Data Structures in PDBLib

The data structures with which the PDBLib system works are passed back to the application program as the result of the high level calls. This gives the programmer access to a great deal of information about the PDB file, the symbol table and the structure chart. Hopefully, this also makes the package more powerful without pushing any special responsibility onto the programmer.

For application developers who require all of the information from a PDB file, the hash lookup function, **SC_def_lookup**, can be used to obtain symbol table entries (**syment**) and structure definitions (**defstr**) from the symbol table or structure chart. Examples can be found in the section at the end of the manual.

8.1 DATA_ALIGNMENT

The set of alignments for the primitive types is kept in a structure called **data_alignment**. Its actual definition is:

```
struct s_data_alignment
   {int char_alignment;
    int ptr_alignment;
    int short_alignment;
    int int_alignment;
    int long_alignment;
    int float_alignment;
    int double_alignment;};

typedef struct s_data_alignment data_alignment;
```

Each member is an integer which specifies the alignment for a primitive type. The alignment is a number of bytes such that a piece of data of this type must begin at an address which is an even multiple of that number. For example, if **ptr_alignment** is 4, a pointer must begin at an address divisible by 4.

8.2 DATA_STANDARD

The set of information describing all of the primitive data types is organized into a structure called a **data_standard**. The **data_standard** characterizes the CPU architecture because all types are either primitive or derived from known types.

Its actual definition is:

```
struct s_data_standard
   {int ptr_bytes;
    int short_bytes;
    int short_order;
    int int_bytes;
    int int_order;
    int long_bytes;
    int long_order;
    int float_bytes;
    long *float_format;
    int *float_order;
    int double_bytes;
    long *double_format;
    int *double_order;};

typedef struct s_data_standard data_standard;
```

The integer types only require a number of bytes and their order. The floating point types require additional information describing the bit layout of the components of the number: the sign bit; the exponent; and the mantissa. These are given as an array of 8 integers as follows:

format[0] = # of bits per number
format[1] = # of bits in exponent
format[2] = # of bits in mantissa
format[3] = start bit of sign
format[4] = start bit of exponent
format[5] = start bit of mantissa
format[6] = high order mantissa bit
format[7] = bias of exponent

8.3 DEFSTR

An entry in the structure chart is represented by a structure called a **defstr**. It contains information about the data type such as the type name, the byte size and alignment, and a list of members.

8.4 DIMDES

A **dimdes** or dimension descriptor contains the information necessary to characterize a list of dimension specifications. It contains such information as the minimum and maximum values the dimension index may have and the net size of the dimension index range.

8.5 MEMDES

A **memdes** or member descriptor is the structure used to contain the information about a member of a **defstr**. It contains information about the type of the member, the name of the member, any dimensions which the member may have, and any casts which have been defined via **PD_cast**.

8.6 PDBFILE

The **PDBfile** is the analog to the **FILE** structure in standard C I/O. In fact, the **PDBfile** contains a **FILE** pointer to access the file via the standard C library functions. In addition, the **PDBfile** contains information such as: the symbol table, the structure charts for the file and the host platform; **data_standard**'s and **data_alignment**'s for the file and the host platform; and a modification date.

8.7 SYMENT

Just as the **defstr** type describes entries in the structure chart the **syment** type describes entries in the symbol table. The **syment** includes information about the data type of the entry, the number of elements, the dimensions of the entry, and its disk address.

9.0 PDBLib by Example

The following code fragments illustrate the functionality of PDBLib. Some of the code is taken from the validation suite and some from the library itself.

9.1 Working with PDB files

This routine is taken from the validation suite for PDBLib. In it, a target for the PDB file is chosen with the routine **test_target** (see the section on **PD_target** for the definition of this function), a PDB file created, some structures defined, data written, and the file closed. The file is then reopened in append mode, some more data written to the file, and the file is closed again. Finally, the file is opened in read mode, the data read, some comparisons done, and the file is closed. The read and write operations are hidden in this example. The significance of the example is that a PDB file is created, closed, and opened in both append and read-only mode.

```
test_1(base, tgt, n)
   char *base, *tgt;
   int n;
   {PDBfile *strm;
    char datfile[MAXLINE], fname[MAXLINE];
    int err;
    FILE *fp;

/* target the file as asked */
    test_target(tgt, base, n, fname, datfile);

    fp = fopen(fname, "w");

/* create the named file */
    if ((strm = PD_open(datfile, "w")) == NULL)
       {fprintf(fp, "Test couldn't create file %s\r\n", datfile);
        exit(1);};
    fprintf(fp, "File %s created\n", datfile);

    prep_test_1_data();

/* make a few defstructs */
    PD_defstr(strm, "l_frame",
              "float x_min", "float x_max", "float y_min",
              "float y_max", LAST);
    PD_defstr(strm, "plot",
              "float x_axis(10)", "float y_axis(10)", "integer npts",
              "char * label", "l_frame view", LAST);

/* write the test data */
```

```
    write_test_1_data(strm);

/* close the file */
    if (!PD_close(strm))
       {fprintf(fp, "Test couldn't close file %s\r\n", datfile);
        exit(1);};
    fprintf(fp, "File %s closed\n", datfile);

/* reopen the file to append */
    if ((strm = PD_open(datfile, "a")) == NULL)
       {fprintf(fp, "Test couldn't open file %s to append\r\n",
                datfile);
        exit(1);};
    fprintf(fp, "File %s opened to append\n", datfile);

    append_test_1_data(strm);

/* close the file after append */
    if (!PD_close(strm))
       {fprintf(fp, "Test couldn't close file %s after append\r\n",
                datfile);
        exit(1);};
    fprintf(fp, "File %s closed after append\n", datfile);

/* reopen the file */
    if ((strm = PD_open(datfile, "r")) == NULL)
       {fprintf(fp, "Test couldn't open file %s\r\n", datfile);
        exit(1);};
    fprintf(fp, "File %s opened\n", datfile);

/* dump the symbol table */
    dump_test_symbol_table(fp, strm->symtab, 1);

/* read the data from the file */
    read_test_1_data(strm);

/* compare the original data with that read in */
    err = compare_test_1_data(strm, fp);

/* close the file */
    if (!PD_close(strm))
```

```
      {fprintf(fp, "Test couldn't close file %s\r\n", datfile);
       exit(1);};
    fprintf(fp, "File %s closed\n", datfile);

/* print it out to stdout */
    print_test_1_data(fp);

    fclose(fp);

    return(err);}
```

9.2 Writing Data to PDB files

These two routines exemplify the various write routines of PDBLib. In particular, they were built to test the spectrum of write calls. Notice their demonstration of the rules for write operations: the variable must be a pointer to data of the type specified.

The identifiers beginning with 'N_' are '#defined' constants whose values are irrelevant to these examples.

```
static char
 cs_w,
 ca_w[N_CHAR],
 *cap_w[N_DOUBLE];

static short
 ss_w,
 sa_w[N_INT];

static int
 is_w,
 ia_w[N_INT],
 p_w[N_INT],
 len;

static float
 fs_w,
 fa2_w[N_FLOAT][N_DOUBLE];

static double
 ds_w,
 da_w[N_FLOAT];

static plot
 graph_w;

static l_frame
 view_w;

static lev1
 *tar_w;
```

```
/*--------------------------------------------------------------------------*/
/*--------------------------------------------------------------------------*/

void write_test_1_data(strm)
   PDBfile *strm;
   {

/* write scalars into the file */
    PD_write(strm, "cs", "char", &cs_w);
    PD_write(strm, "ss", "short", &ss_w);
    PD_write(strm, "is", "integer", &is_w);
    PD_write(strm, "fs", "float", &fs_w);
    PD_write(strm, "ds", "double", &ds_w);

/* write primitive arrays into the file */
    PD_write(strm, "sa(5)", "short", sa_w);
    PD_write(strm, "ia(5)", "integer", ia_w);

/* write structures into the file */
    PD_write(strm, "view", "l_frame", &view_w);
    PD_write(strm, "graph", "plot", &graph_w);

    return;}

/*--------------------------------------------------------------------------*/
/*--------------------------------------------------------------------------*/

void write_test_2_data(strm)
   PDBfile *strm;
   {

   PD_write(strm, "tar", "lev1 *", &tar_w);

    return;}

/*--------------------------------------------------------------------------*/
/*--------------------------------------------------------------------------*/
```

9.3 Reading Data from PDB files

These two routines exemplify the various read routines of PDBLib. In particular, they were built to test the spectrum of read operations. They read the data written out in the previous example. Notice their demonstration of the rules for read operations: the variable must be a pointer to data of the type specified. This is often a more difficult proposition for reads since the type information isn't supplied in the read call.

The member read operations at the end of the first routine should be studied carefully. They are not the most general read example, but they are among the most useful.

The second routine reads not only the entire structure, but picks out each part individually. It demonstrates the rule about dereferencing pointers in the partial read operation. Study the structures of this example carefully!

The identifiers beginning with 'N_' are '#defined' constants whose values are irrelevant to these examples. Also, take for granted that the unspecified variables to contain parts of the structures have the correct declarations.

```
struct s_lev2
 {char **s;
 int type;};

typedef struct s_lev2 lev2;

struct s_lev1
 {int *a;
 double *b;
 lev2 *c;};

typedef struct s_lev1 lev1;

static char
 cs_r,
 ca_r[N_CHAR],
 *cap_r[N_DOUBLE];

static short
 ss_r,
 sa_r[N_INT];

static int
 is_r,
```

```
 ia_r[N_INT];

static float
 fs_r,
 fs_app_r,
 fs_p1_r,
 fs_p2_r,
 fs_p3_r,
 fa2_r[N_FLOAT][N_DOUBLE],
 fa2_app_r[N_FLOAT][N_DOUBLE];

static double
 ds_r,
 da_r[N_FLOAT];

static plot
 graph_r;

static l_frame
 view_r;

static lev1
 *tar_r;

/*--------------------------------------------------------------------------*/
/*--------------------------------------------------------------------------*/

void read_test_1_data(strm)
   PDBfile *strm;
   {

/* read the scalar data from the file */
    PD_read(strm, "cs", &cs_r);
    PD_read(strm, "ss", &ss_r);
    PD_read(strm, "is", &is_r);
    PD_read(strm, "fs", &fs_r);
    PD_read(strm, "ds", &ds_r);

/* read the primitive arrays from the file */
    PD_read(strm, "ca", ca_r);
    PD_read(strm, "sa", sa_r);
```

```
    PD_read(strm, “ia”, ia_r);
    PD_read(strm, “fa2”, fa2_r);
    PD_read(strm, “da”, da_r);
    PD_read(strm, “cap”, cap_r);

/* read the entire structures from the file */
    PD_read(strm, “view”, &view_r);
    PD_read(strm, “graph”, &graph_r);

/* read the appended data from the file */
    PD_read(strm, “fs_app”, &fs_app_r);
    PD_read(strm, “fa2_app”, fa2_app_r);

/* struct member test */
    PD_read(strm, “graph.view.x_max”, &fs_p2_r);
    PD_read(strm, “view.y_max”, &fs_p3_r);

    return;}

/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/

/* READ_TEST_2_DATA - read the test data from the file */

void read_test_2_data(strm)
   PDBfile *strm;
   {do_r = strm->default_offset;

    PD_read(strm, “tar”, &tar_r);
    PD_read(strm, “p”, p_r);

    PD_read(strm, “tar(0).a”, &ap1);
    PD_read(strm, “tar(1).a”, &ap2);

    PD_read(strm, “tar(0).a(0)”, &aa[0]);
    PD_read(strm, “tar(0).a(1)”, &aa[1]);
    PD_read(strm, “tar(1).a(0)”, &aa[2]);
    PD_read(strm, “tar(1).a(1)”, &aa[3]);

    PD_read(strm, “tar(0).b”, &bp1);
    PD_read(strm, “tar(1).b”, &bp2);
```

```
PD_read(strm, “tar(0).b(0)”, &ba[0]);
PD_read(strm, “tar(0).b(1)”, &ba[1]);
PD_read(strm, “tar(1).b(0)”, &ba[2]);
PD_read(strm, “tar(1).b(1)”, &ba[3]);

PD_read(strm, “tar(0).c”, &cp1);
PD_read(strm, “tar(1).c”, &cp2);

PD_read(strm, “tar(0).c(0)”, &ca[0]);
PD_read(strm, “tar(0).c(1)”, &ca[1]);
PD_read(strm, “tar(1).c(0)”, &ca[2]);
PD_read(strm, “tar(1).c(1)”, &ca[3]);

PD_read(strm, “tar(0).c(0).s”, &sp1);
PD_read(strm, “tar(0).c(1).s”, &sp2);
PD_read(strm, “tar(1).c(0).s”, &sp3);
PD_read(strm, “tar(1).c(1).s”, &sp4);

PD_read(strm, “tar(0).c(0).s(0)”, &tp1);
PD_read(strm, “tar(0).c(0).s(1)”, &tp2);
PD_read(strm, “tar(0).c(1).s(0)”, &tp3);
PD_read(strm, “tar(0).c(1).s(1)”, &tp4);

PD_read(strm, “tar(0).c(0).s(0)(2)”, &ta[0]);
PD_read(strm, “tar(0).c(0).s(1)(1)”, &ta[1]);
PD_read(strm, “tar(0).c(1).s(0)(3)”, &ta[2]);
PD_read(strm, “tar(0).c(1).s(1)(2)”, &ta[3]);

PD_read(strm, “tar(1).c(0).s(0)”, &tp5);
PD_read(strm, “tar(1).c(0).s(1)”, &tp6);
PD_read(strm, “tar(1).c(1).s(0)”, &tp7);
PD_read(strm, “tar(1).c(1).s(1)”, &tp8);

return;}
```

9.4 Inquiries in PDBLib

The following fragments show how to obtain information about PDB files and their contents.

```
/*--------------------------------------------------------------*/
/*--------------------------------------------------------------*/

myread(file, name, var, offset, number)
   PDBfile *file;
   char *name;
   void *var;
   long offset, number;
   {long addr, num;
    char *token, *type, memb[MAXLINE];
    dimdes *dims;
    syment *ep;

    strcpy(memb, name);
    token = strtok(memb, ".([");

/* look up the variable name */
    ep = PD_inquire_entry(file, token, false, NULL);
    if (ep == NULL)
        PD_error("VARIABLE NOT IN SYMBOL TABLE - MYREAD",
                 READ);

    addr = PD_entry_address(ep);
    dims = PD_entry_dimensions(ep);
    num  = PD_entry_number(ep);
    type = PD_entry_type(ep);

/* with ep in hand, we know the variable type, number of elements,
 * dimensions, and disk address
 */
              .
              .
              .

/*--------------------------------------------------------------*/
/*--------------------------------------------------------------*/
```

```
_PD_wr_leaf(file, var, nitems, type)
   PDBfile *file;
   char *var;
   long nitems;
   char *type;
   {char *svar;
    int size;
    FILE *fp;
    defstr *dp;
    memdes *desc, *mem_lst;

    fp = file->stream;
           .
           .
           .

/* dispatch all other writes */
    if (file->conversions)
           .
           .
           .

/* obtain a pointer to the defstr associated with type */
    dp = PD_inquire_host_type(file, type);
    if (dp == NULL)
       PD_error("BAD TYPE - WR-LEAF", WRITE);

/* if the structure has any pointered members loop over the members */
    if (dp->n_indirects && ((mem_lst = dp->members) != NULL))
       {size = dp->size;
        for (svar = var, offset = 0L, i = 0L; i < nitems; i++)
            {for (desc = mem_lst; desc != NULL; desc = desc->next)
                 { ... };
             svar += size;};};
           .
           .
           .

/*----------------------------------------------------------------*/
/*----------------------------------------------------------------*/
```

9.5 Writing PM_mappings with PFWMAP

The following fragment shows how to write a **PM_mapping** to a PDB file from a FORTRAN program. This example uses **PFWMAP**. **PFWMAP** writes out both domain and range as a complete mapping. The drawback to this approach is that many mappings might have the same domain which would be written out over and over again. This could lead to unacceptably large data files. See the next example for a more space saving approach.

```
c
c**************************************************************
c**************************************************************
c
      subroutine sxdmp(namesx, namelen)

c traverse plot list to make dumps suitable for sx

      integer pfopen, pfclos, pfgerr, pfwmap, pfwset, pfwran
      integer zscanleq

      integer pderr(32), rp(6), dp(6)
      integer fileid, pim, dmnsn

      double precision rm(nsavept), dm(nsavept)

c ... set up
            .
            .
            .

c ... open the file
      fileid = pfopen(namelen, namesx, 'w')
      if (fileid .eq. -1) then
         iplpderr = pfgerr(nchr, pderr)
         call errproc(pderr, nchr, 0, 0)
      endif

      pim = 0

c ... loop over plots
      do ...

c ... setup domain
```

```
 dname = <domain name>
 dp(1) = <length of domain name>

 select case (dmnsn)
 case ('d2r1')
    dp(2) = 2
    dp(3) = 2
    dp(4) = nr
    dp(5) = kmax
    dp(6) = lmax
    call zmovewrd(dm(1), x2d, nr)
    call zmovewrd(dm(nr+1), y2d, nr)
 case ('d1r1')
    dp(2) = 1
    dp(3) = 1
    dp(4) = nr
    dp(5) = nr
    call zmovewrd(dm(1), <domain data>, nr)
 endselect

c ... set up range
 rname = <range name>
 rp(1) = <length of range name>

 rp(2) = dp(2)
 rp(3) = 1
 rp(4) = nr
 rp(5) = dp(5)
 rp(6) = dp(6)
 call zmovewrd(rm(1), <range data>, nr)

c ... write the mapping
 ierr = pfwmap(fileid, dname, dp, dm, rname, rp, rm, pim)
 if (ierr.eq.0) then
    pfgerr(nchr, pderr)
    call errproc(pderr, nchr, 0, 0)
 endif
 pim = pim + 1
 repeat

c ... close the file
```

```
icloseok = pfclos(fileid)
if (icloseok.eq.0) then
   pfgerr(nchr, pderr)
   call errproc(pderr, nchr, 0, 0)
endif

return
end
```

9.6 Writing PM_mappings with PFWSET and PFWRAN

The following fragment shows how to write a **PM_mapping** to a PDB file from a FORTRAN program. This example uses **PFWSET** and **PFWRAN**. With **PFWSET** and **PFWRAN** an application can write mappings in the most space efficient way. Instead of writing the same domains over and over again as would be done with **PFWMAP**, the application can select the unique domains and write them out with **PFWSET**. Then all mappings can be written using **PFWRAN** which writes out a **PM_mapping** with a null domain. The post processing tools (such as PDBView) reconstruct the complete **PM_mapping** by looking for the domain as a variable with the same name as the domain component of the mapping name. This approach is clearly a little more involved than using **PFWMAP** and application developers should weigh the advantages and disadvantages before selecting one method over the other. See the previous example for an illustration of the use of **PFWMAP**.

```
c
c*************************************************************
c*************************************************************
c
 subroutine sxdmp(namesx, namelen)

c traverse plot list to make dumps suitable for sx

 integer pfopen, pfclos, pfgerr, pfwmap, pfwset, pfwran
 integer zscanleq

 integer pderr(32), rp(6), dp(6)
 integer fileid, pim, dmnsn

 double precision rm(nsavept), dm(nsavept)

c ... set up
            .
            .
            .

c ... open the file
 fileid = pfopen(namelen, namesx, 'w')
 if (fileid .eq. -1) then
    iplpderr = pfgerr(nchr, pderr)
    call errproc(pderr, nchr, 0, 0)
 endif
```

```
 pim = 0

c ... loop over plots
 do ...

c ... setup domain
 dname = <domain name>
 dp(1) = <length of domain name>

 select case (dmnsn)
 case ('d2r1')
    dp(2) = 2
    dp(3) = 2
    dp(4) = nr
    dp(5) = kmax
    dp(6) = lmax
    call zmovewrd(dm(1), x2d, nr)
    call zmovewrd(dm(nr+1), y2d, nr)
 case ('d1r1')
    dp(2) = 1
    dp(3) = 1
    dp(4) = nr
    dp(5) = nr
    call zmovewrd(dm(1), <domain data>, nr)
 endselect

 if <unique domain> then
    ierr = pfwset(fileid, dname, dp, dm)
    if (ierr.eq.0) then
       iplpderr = pfgerr(nchr, pderr)
       call errproc(pderr, nchr, 0, 0)
    endif
 endif

c ... set up range
 rname = <range name>
 rp(1) = <length of range name>

 rp(2) = dp(2)
 rp(3) = 1
 rp(4) = nr
```

```
 rp(5) = dp(5)
 rp(6) = dp(6)
 call zmovewrd(rm(1), <range data>, nr)

c ... write out range
 ierr = pfwran(fileid, dname, dp(1), rname, rp, rm, pim)
 if (ierr.eq.0) then
    pfgerr(nchr, pderr)
    call errproc(pderr, nchr, 0, 0)
 endif
 pim = pim+1
 repeat

c ... close the file
 icloseok = pfclos(fileid)
 if (icloseok.eq.0) then
    pfgerr(nchr, pderr)
    call errproc(pderr, nchr, 0, 0)
 endif

 return
 end
```

10.0 Related Documentation

PDBLib is one part of a collection of libraries called PACT. PDBLib uses the SCORE library in PACT for memory management, hash table, and string handling support. Interested readers should consult the SCORE manual as well as the PDBView manual, the ULTRA II manual, and the PANACEA manual for more information on how PDBLib is used.

The list of PACT documents is:

PACT User's Guide, UCRL-MA-112087

SCORE User's Manual, UCRL-MA-108976 Rev.1

PPC User's Manual UCRL-MA-108964 Rev.1

PML User's Manual, UCRL-MA-108965 Rev.1

PDBLib User's Manual, M-270 Rev.2 (this document)

PGS User's Manual, UCRL-MA-108966 Rev.1

PANACEA User's Manual, M-276 Rev.2

ULTRA II User's Manual, UCRL-MA-108967 Rev.1

PDBDiff User's Manual, UCRL-MA-108975 Rev.1

PDBView User's Manual, UCRL-MA-108968 Rev.1

SX User's Manual, UCRL-MA-112315

L

M

N

O

P