Portable Self-Describing Binary Data Streams

Greg Eisenhauer
eisen@cc.gatech.edu

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332

August 27, 1997 – PBIO Version 3.1

Abstract

Storing and transmitting data in binary form is often desirable both to conserve I/O bandwidth and to reduce storage and processing requirements. However, transmission of binary data between machines in heterogeneous environments is problematic. When binary formats are used for long-term data storage, similar problems are encountered with data portability. This paper presents the application that led us to work on this problem, evaluates other standards for binary files and discusses our solution.

1 Introduction

The need for binary I/O is often encountered in situations where I/O speed must be maximized. Yet, while tools for the processing and manipulation of text files and data streams are common, those for their binary counterparts are few. Given the trend toward heterogeneous, highly-networked computing environments, the need for better approaches in binary I/O become even more apparent.

Our own needs for binary data streams derive from our work in parallel program monitoring and steering. The program steering environment demanded the speed and compactness of binary data transmission in a heterogeneous computing environment. The requirement to support both on-line and post-execution processing of monitoring data led to requirements for both stream and file support. Relatively independent development of the monitoring, steering and display systems indicated a need for robustness and correctness checking in the data exchange. These considerations led us to define the following characteristics as important for binary files and data streams:

Portability – Data stored in the format should be portable across machines despite differences in byte order, default integer size, etc. If native and "file" data formats differ, those differences should be hidden as much as possible without loss of data.

Low Overhead – The format should not impose significantly more overhead than raw binary I/O. For example, the format should not require data to be translated into a "standard" or "network" representation, which would require data translation on both input and output for some machines.

Stream Interpretation – The format should be usable both as a file format and for a data stream such as might come from a socket. This implies that the format must be interpretable on-the-fly without requiring the entire data set to be present.

Robustness – A major difficulty with using raw binary I/O is that minor changes in the output program, such as adding a field or record, require exact corresponding changes in all input programs. Making such lock-step changes is an exacting process and the bugs produced by errors are difficult to find. Any such changes also usually invalidate all existing data files. Data may be lost simply because the exact sequence used to write it is no longer known. Any new approach should resolve these problems.

Tool Generation – The format should contain enough meta-information to enable the creation of tools that operate on data without compiled-in knowledge of its nature.

Ease of Use – The format should support easy description and I/O of basic data elements such as integers, floats and strings, as well as nested structures and simple arrays of these elements.

2 Other Approaches

Many scientific communities have established binary data formats oriented toward their specific needs. However from the 'robustness' and 'tool generation' requirements above it was clear that a simple format would not necessarily fulfill our needs. Instead we needed a *meta-format* in which the actual formats of binary records could be described. We examined several existing meta-formats to see if they met our requirements.

HDF[2] and netCDF[3] are file meta-formats designed for use by the general scientific community. Both contain extensive support for data description and both address portability, though not to the extent we require. They are also oriented towards long-term data storage and data exchange and so do not directly address issues of stream interpretation.

The Pablo Self-Defining Data Format (SDDF)[1] is a meta-format designed to support monitoring trace files with requirements similar to our own. However SDDF's presumption that the sizes of basic data types do not change causes portability problems. While SDDF has a binary representation, its ASCII representation must be used for true portability. Also, though SDDF's C++ support of fetching fields individually provides some measure of robustness, it will also be considerably slower than raw binary I/O.

Given that existing meta-formats failed to meet our requirements we chose to develop our own. Because we had little need for representing such things as images and hyperslabs we concentrated less on abstract data representation and more on ease of use, portability and robustness in the face of changing data. Unlike SDDF, our meta-format is designed as a black-box. The user sees only the library routines and the actual representation of meta-data is hidden. The next section describes the nature of our meta-format and the library that supports its use.

3 Basic I/O using the PBIO Library

The basic approach of the Portable Binary I/O library is relatively simple. PBIO files are record-oriented. Writers of data must provide a description of the names, types, sizes and positions of the fields in the records they are writing. Readers must provide similar information for the records that they are interested in reading. No translation is done on the writer's end. On the reader's end, the format of the incoming record is compared with the format that the program expects. If there are discrepancies the PBIO read routine performs the appropriate translations.

3.1 Supported Data Types

The PBIO routines support record formats consisting of fields of the following basic data types: "integer", "unsigned integer", "float", "char", "enumeration" and "string". Note that *field type* here is separate from *field size* so that both the native C types "int" and "long" are "integer" types. "char" is essentially treated as a small "integer" except that the **IOdump** program will print it as a character. "enumeration" is also treated as an integer and there is currently no mechanism to apprise the IO routines of the symbolic names associated with the values. "string" is a C-style zero-terminated variant-length string. Both NULL and zero-length strings are handled appropriately.

There is no prohibition on the use of types not listed here. However translation and display are not available for other than the built-in types.

3.2 Field Lists

A record format is characterized by a field list. Each field is specified by its name, basic data type, the field's size and offset in the record. The field name and basic data type are specified with strings. The size

and offset are integers. Below is an example structure for a record and the corresponding field list:

```
typedef struct _first_rec {
    int         i;
    long        j;
    double      d;
    char        c;
} first_rec, *first_rec_ptr;

static IOField field_list[] = {
    {"i", "integer", sizeof(int), IOOffset(first_rec_ptr, i)},
    {"j", "integer", sizeof(long), IOOffset(first_rec_ptr, j)},
    {"d", "float",   sizeof(double), IOOffset(first_rec_ptr, d)},
    {"c", "integer", sizeof(char), IOOffset(first_rec_ptr, c)},
    {NULL, NULL, 0, 0},
};
```

The "IOOffset" macro simply calculates the offset of a field in a structure using compile-time information. Its use is recommended to avoid hand-calculating and hard-coding offsets. The order of fields in the field list is not important. It is not even necessary to specify every field in a structure with an entry in the field list. Unspecified fields at the end of the structure may not be written to the IO file. Unspecified fields at the beginning or in the middle of a structure will be written to the IO file, but no information about them will be available.

3.3 Formats and Conversions

While field lists characterize the layout of records, it would be inefficient to process the string-oriented field list on every record read or write. To avoid this inefficiency, field lists must be analyzed prior to reading or writing data. For output, field lists are "registered" with a particular output file with the call *register_IOrecord_format*(). This call specifies a name to be associated with the record format in the file and returns a handle, of the type *IOFormat*. The returned *IOFormat* is used in the *write_IOfile* call and specifies the format of the data to be written to the file. The names of record formats must be unique within a PBIO file and are used on the reading end to identify specific record format within the file. Because there is no translation upon write in the PBIO scheme, the field list which governs the writing *IOFormat* becomes the *file record format* in the written file.

In the case of reading a PBIO file, IOConversions facilitate the common case where the reading program knows, for the records in which it interested, the names of both the record format and the fields within those format which it requires. For reading, the subroutine *set_IOconversion*() serves a similar function as *register_IOrecord_format*(). However, instead of the field list specifying the format of the records in the file, it specifies the fields required *from* the file and the program format into which they are to be converted. The format name specified in the *set_IOconversion*() call must match the name of a format in the file. The record format in the file must contain *at least* all the fields specified in the conversion field list. If there are more fields in the file record format than the reader specifies in the conversion, those additional fields in file records are ignored. For the fields that are common between the formats, the PBIO library will perform any data rearrangement or conversion that is required to convert the incoming binary file record into a native record. Reading programs *must* set a conversion for incoming data they wish to have converted into comprehensible form.

3.4 Simple Read and Write Programs

Given the structure and field declarations above, a simple writer and reader programs are shown in figures 1 and 2. These simple programs handle the most common data transfer situation.[1] The source program supplies the complete specification of the data being written. The destination program specifies the fields

[1] Both the sample reader and writer programs use the routine *open_IOfile*(), which is used for file-based data exchanges. The routine *open_IOfd*() takes an integer file descriptor instead of a filename as an argument and is used for socket- or stream-based exchanges.

```
int main()
{
    IOFile iofile = open_IOfile("test_output", "w");
    IOFormat first_rec_ioformat;
    first_rec rec1;
    int i;

    first_rec_ioformat = register_IOrecord_format("first format",
                                                  field_list,
                                                  iofile);
    for(i=0; i<10; i++) {
        rec1.i = i;  rec1.j = 2*i; rec1.d = 2.727 + i; rec1.c = 'A' + 2*i;
        if(!write_IOfile(iofile, first_rec_ioformat, &rec1)) {
           printf("write failed\n");
        }
    }
    close_IOfile(iofile);
}
```

Figure 1: A simple writer program.

that it is interested in and their locations and sizes in the data structure in which it would like them placed. Fields are matched by name. If there are differences in structure specifications, the PBIO routines perform the appropriate conversions at the time of the read operation. The reader program will read the binary files produced by the writer program, despite potential difference in:

- byte ordering on the reading and writing architectures.
- differences in sizes of datatypes (e.g. long and int).
- compiler structure layout differences.

In the general case, the reading and writing program need not be using the same structures at all, although all the fields that the reading program specifies *must* actually exist in the data. The IO routines can also convert field a float field to an integer and vice versa. There is, of course, the possibility of data loss in any conversion. If the user requests that an 8-byte integer in the data stream be placed in a 2-byte integer in the program, the top 6 bytes will be lost. A floating point value may not be precisely representable as an integer nor a large integer by a floating point value. At present, loss of data occurs quietly, but future extensions

```
void
main(argc, argv)
int argc;
char **argv;
{
    IOFile iofile = open_IOfile("test_output", "r");
    first_rec rec1;
    int i;

    set_IOconversion(iofile, "first format", field_list, sizeof(first_rec));
    for(i=0; i<10; i++) {
        read_IOfile(iofile, &rec1);
        printf("rec had %d, %d, %g, %c\n", rec1.i, rec1.j, rec1.d, rec1.c);
    }
    close(iofile);
    exit(0);
}
```

Figure 2: A simple reader program.

may address this issue. Note that the reading program must set a conversion for all record formats for which it wishes to use $read\_IOfile()$. If no conversion is set, the $read\_IOfile()$ call will fail and the record will be discarded.

4 More Complex Issues

The programs in the presented previous section are sufficient to handle the transmission of simple atomic data types in the simplest of circumstances. This section will expand on that implementation basis with additional facilities.

4.1 Strings and Memory Handling

The sample programs presented above exercise all the basic data types including "string", but it leaves open some questions about memory management. The principal complication in handling variable-length strings is that exact storage requirements aren't known beforehand. This isn't an issue on the writing end, but on the reading end either the user must provide memory for strings or it must be allocated by the PBIO library. It is possible to use either approach with the PBIO library. In particular, if records containing string datatypes are read using the $read\_IOfile()$ routine, memory for the strings is allocated in a temporary buffer internal to the PBIO library and the $char*$ fields in the record given to the user point into this internal buffer. There is one buffer per IOfile and its contents remain valid only until the next PBIO call. In this circumstance, it is the users responsibility to ensure that $char*$ pointers in input records are only used while buffer contents are still valid.

A program which requires direct control of string memory should use the routines $next\_IOrecord\_length()$ and $read\_to\_buffer\_IOfile()$. $next\_IOrecord\_length()$ returns the number of bytes required to contain the entire contents of the next record. This returned size is the size of the native record structure plus the length of each string in the incoming record (counting the terminating zero byte). $read\_to\_buffer\_IOfile()$ reads the input record and all its associated strings into an appropriately size buffer. The record is placed at the beginning of the buffer and it is immediately followed by the string contents. The actual $char*$ value fields in the record are pointer to the strings later in the buffer. Thus the whole record including the strings it contains can be treated as a unit for memory allocation. Figure 3 shows total memory requirements for an example buffer layout which might result from reading a record containing a string.

4.2 Complex Data Types

PBIO also offers some facilities for constructing records which consist of more than just the simple data types described in Section 3.1. The simplest is a mechanism for declaring a field in a record to be an array of atomic data types. For example, a type specification of "integer[5]" is understood to be an single dimensional array of 5 integers. "float[3][3]" is a 3×3 two dimensional array floating point numbers. At present, PBIO supports these fixed array sizes with one or two dimensions. The field size specified should be the size of a single array element, **not** the size of entire array. (This convention was different in earlier versions of this

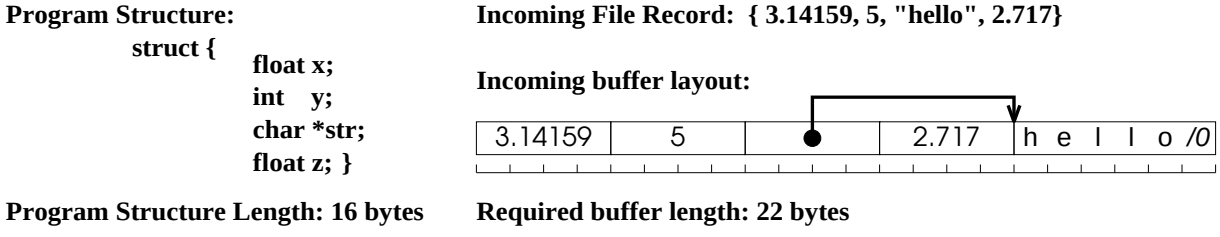

Figure 3: Buffer layout for an incoming record containing a string.

library.) When reading a record containing array fields, the dimensionality of each field must match that which was used when the record was written, though the size of the elements may differ.

In addition to fixed array sizes, PBIO supports dynamically sized one-dimensional arrays. In this case, the size in the array type specification must be the string name of an integer field in the record. The value of that integer field gives the array size. The actual data type in the record should be a pointer to the element type. Figure 4 gives an example of a dynamic array declaration. For the purposes of memory allocation (as discussed in Section 4.1), the dynamic arrays are treated like strings on reads. That is, *read_IOfile*() leaves the arrays in temporary memory that will remain valid until the next PBIO operation and *read_to_buffer_IOfile*() copies the arrays into a user-supplied buffer. In the case of a record which contains a dynamic array, *next_IOrecord_length*() still returns the number of bytes required to hold the entire record including the memory required for the dynamic array.

```
typedef struct _dyn_rec {
    char        *string;
    long        icount;
    double      *double_array;
} dyn_rec, *dyn_rec_ptr;

IOField dyn_field_list[] = {
    {"string field", "string", sizeof(char *),
      IOOffset(dyn_rec_ptr, string)},
    {"icount", "integer", sizeof(long),
      IOOffset(dyn_rec_ptr, icount)},
    {"double_array", "float[icount]", sizeof(double),
      IOOffset(dyn_rec_ptr, double_array)},
    { NULL, NULL, 0, 0}
};
```

Figure 4: A dynamic array record format

Finally, a field type may be the name of a previously registered record format. This facility can be used to define record formats in a hierarchical way. For example, the structure `particle_struct` declared as in Figure 5 could be written by registering formats defined by these field lists:

```
typedef struct R3vector_struct {
    double x, y, z;
} R3vector;

typedef struct particle_struct {
    R3vector   loc;
    R3vector   deriv1;
    R3vector   deriv2;
} particle;
```

Figure 5: A nested record format

```
static IOField R3field_list[] = {
    {"x", "float", sizeof(double), IOOffset(R3vector*, x)},
    {"y", "float", sizeof(double), IOOffset(R3vector*, y)},
    {"z", "float", sizeof(double), IOOffset(R3vector*, z)},
    {NULL, NULL, 0, 0},
};

static IOField particle_field_list[] = {
    {"loc", "R3vector", sizeof(R3vector), IOOffset(particle*, loc)},
    {"deriv1", "R3vector", sizeof(R3vector), IOOffset(particle*, deriv1)},
    {"deriv2", "R3vector", sizeof(R3vector), IOOffset(particle*, deriv2)},
    {NULL, NULL, 0, 0},
};
```

4.3 Formats and Record Types

The programs in Section 3.4 are simplistic in that only known number records of a single type are written and read. When multiple formats or unknown numbers of records are involved, the reading program needs to know what, if anything, is coming next. PBIO allows access to this information via the *next_IOrecord_format*() call. This subroutine returns a value of the type *IOFormat*. If this value is NULL, an end of file or error condition has been encountered. If non-NULL, the value can be passed to the subroutine *name_of_IOformat*() to get the string name associated with the format of the next data record. Additionally, for programs which wish to avoid multiple string comparison operations on every read operation, PBIO provides the *get_IOformat_by_name*() subroutine. With these operations, the simple reader and writer programs of Section 3.4 can be rewritten to handle records written in any order. Figures 6 and 7 give the main bodies of these programs and assume the structure and field list definitions used earlier in this paper.

The programs in Section 3.4 are also simplistic in that the writer registers all record formats before writing any data records and the reader will not work if this condition is violated. Many simple programs

```
first_rec_ioformat = register_IOrecord_format("first format", field_list, iofile);
vec_ioformat = register_IOrecord_format("R3vector", R3field_list, iofile);
particle_ioformat = register_IOrecord_format("particle", particle_field_list, iofile);
srandom(time(NULL));
strcpy(str, "A String");
rec1.s = str;
for(i=0; i<10; i++) {
    if (random() % 2 == 1) {
        first_rec rec1;
        rec1.i = i;  rec1.j = 2*i; rec1.d = 2.727 + i; rec1.c = 'A' + 2*i;
        strcat(str, "!");
        if(!write_IOfile(iofile, first_rec_ioformat, &rec1)) {
            printf("write failed\n");
        }
    } else {
        particle p;
        double s = i * i;
        double c = s * i;
        p.deriv2.x = 3.0*i;  p.deriv2.y = 4.2*i;  p.deriv2.z = 4.8*i;
        p.deriv1.x = 1.5*s;  p.deriv1.y = 2.1*s;  p.deriv1.z = 2.4*s;
        p.loc.x    =  .5*c;  p.loc.y    =  .7*c;  p.loc.z    =  .8*c;
        if(!write_IOfile(iofile, particle_ioformat, &p)) {
            printf("write failed\n");
        }
    }
}
```

Figure 6: The body of a more complex writer program.

```
IOFile iofile = open_IOfile("test_output", "r");
IOFormat first_format, particle_format, next_format;

set_IOconversion(iofile, "first format", field_list, sizeof(first_rec));
set_IOconversion(iofile, "R3vector", R3field_list, sizeof(R3vector));
set_IOconversion(iofile, "particle", particle_field_list, sizeof(particle));

first_format = get_IOformat_by_name(iofile, "first format");
particle_format = get_IOformat_by_name(iofile, "particle");

next_format = next_IOrecord_format(iofile);
while(next_format != NULL) {
    if (next_format == first_format) {
        first_rec rec1;
        read_IOfile(iofile, &rec1);
        printf("rec had %d, %d, %g, %s, %c\n", rec1.i, rec1.j, rec1.d,
               rec1.s, rec1.c);
    } else if (next_format == particle_format) {
        particle p;
        read_IOfile(iofile, &p);
        printf("particle.loc = %g, %g, %g, deriv1 = %g, %g, %g\n", p.loc.x,
               p.loc.y, p.loc.z, p.deriv1.x, p.deriv1.y, p.deriv1.z);
    }
    next_format = next_IOrecord_format(iofile);
}
```

Figure 7: The body of a more complex reader program.

use a fairly static set of record formats for I/O and have no problems registering all formats at the beginning. But in some circumstances, a program may need to add a new record format at a later time. Others may even need to change the layout or sizes of format fields on the fly. For the writer, this isn't a significant problem. PBIO allows new record formats to be registered to an output stream at any time. However, reading programs need a way of knowing when new formats are encountered on input.

In the PBIO library, data records are just one of the types of records which appear in the input stream. Data records are of principal interest to many programs so the PBIO interface is designed to make access to those records easy. However, format descriptions are implicitly written to output streams whenever a new record format is registered. In the simple programs presented thus far, record formats are read implicitly when encountered by the data input routines. However, those descriptions are available for reading if so desired and constitute a second record type. The current version of PBIO also allows *comments* to be embedded in the data stream. Comments are simple null-terminated strings that are not interpreted by the PBIO routines but are available for "labeling" files or data streams. Comments are written with the $write\_IOcomment()$ function and are the third type of record which can appear in a PBIO input stream. The function $next\_IOrecord\_type()$ returns the type of the next record in an input stream. Its return value is one of an enumeration consisting of the values $\{IOdata, IOformat, IOcomment, IOend,$ and $IOerror\}$.

Figure 8 shows the body of a reader program that is capable of handling new formats at any time. Its organization is somewhat different from the previous reader program of Figure 7. For example, the new program is careful to set conversions for formats only after they have been read. Trying to set a conversion for a record format which has not yet been seen is an error. The new program also demonstrates how to handle comments and unwanted records in the input stream. In the case of comments, the comment string is held in a buffer internal to PBIO and the $read\_comment\_IOfile()$ call returns a pointer to this buffer. The buffer contents are only guaranteed valid until the next PBIO operation. Unwanted buffers are discarded by issuing a $read\_IOfile()$ call with a NULL buffer address. This has the effect of consuming the next buffer on the input stream.

4.4 Bulk Record Handling

PBIO offers some limited facilities for handling more than one data record at a time. These facilities can be separated into two groups, one intended to support handling contiguous blocks of records and the other for writing smaller numbers of records.

The support for contiguous blocks of records is provided by the routines:

```
extern int
read_array_IOfile(IOFile iofile, void *data, int count, int struct_size);

extern int
write_array_IOfile(IOFile iofile, IOFormat ioformat, void *data, int count, int struct_size);
```

$write\_array\_IOfile()$ writes *count* records of the same format to the IOfile. *data* points to the start of the block of records and $struct_size$ must be the size of each array element. Note that the size of the array element may be different than the size of the structure outside of the array because of compiler and data alignment issues. This type of array write operation is only available for record formats which do not contain any fields of type `string`. All records in the array are written with a single $write()$ system call.

On the reading side, $read\_array\_IOfile()$ performs a similar function. Records which have been written as arrays can be read individually with the normal $read_IOfile()$ routine. However, it is not possible to read as an array records which have not been written with $write\_array\_IOfile()$. $read\_array\_IOfile()$ returns the number of records which were read, up to *count*. All the records will be read with a single $read()$ system call. The routine $next_IOrecord_count()$ returns the number of array records pending.[2]

[2] "array records" are records which have been written with $write\_array\_IOfile()$. The number pending is the number remaining in the current set that were written in a single call.

```
IOFile iofile = open_IOfile("test_output", "r");
IOFormat first_format, particle_format, next_format;

while(1) {
    switch(next_IOrecord_type(iofile)) {
    case IOend:
    case IOerror:
        close(iofile);
        exit(0);
        break;
    case IOformat:
        next_format = read_format_IOfile(iofile);
        if (strcmp("first format", name_of_IOformat(next_format)) == 0) {
            first_format = next_format;
            set_IOconversion(iofile, "first format", field_list, sizeof(first_rec));
        } else if (strcmp("particle", name_of_IOformat(next_format)) == 0) {
            particle_format = next_format;
            set_IOconversion(iofile, "particle", particle_field_list, sizeof(particle));
        } else if (strcmp("R3vector", name_of_IOformat(next_format)) == 0) {
            set_IOconversion(iofile, "R3vector", R3field_list, sizeof(R3vector));
        } else {
            /* no need to track other formats */
        }
        break;
    case IOdata:
        next_format = next_IOrecord_format(iofile);
        if (next_format == first_format) {
            first_rec rec1;
            read_IOfile(iofile, &rec1);
            printf("rec had %d, %d, %g, %s, %c\n", rec1.i, rec1.j, rec1.d,
                   rec1.s, rec1.c);
        } else if (next_format == particle_format) {
            particle p;
            read_IOfile(iofile, &p);
            printf("particle.loc = %g, %g, %g, deriv1 = %g, %g, %g\n", p.loc.x,
                   p.loc.y, p.loc.z, p.deriv1.x, p.deriv1.y, p.deriv1.z);
        } else {
            /* read and discard other records */
            read_IOfile(iofile, NULL);
        }
        break;
    case IOcomment:
        {
            char *comment = read_comment_IOfile(iofile);
            printf("Got comment >%s<\n", comment);
            break;
        }
    }
}
```

Figure 8: A reader program for dynamic format registration.

The other facility for bulk record handling is the routine *writev_IOfile*(). *writev_IOfile*() is similar to the *writev*() system call. Instead of taking single data buffer (of a particular format) to write, *writev_IOfile*() takes a list of data buffers and formats. To the extent possible, all these buffers will be written to the output stream with a single *writev*() system call.[3] *writev_IOfile*() imposes no restrictions on the nature of the fields in the records to be written. Unfortunately, the nature of the PBIO protocol allows no corresponding read call. Records written a single *writev_IOfile*() must be read with multiple *read_IOfile*() calls. The prototype of *writev_IOfile*() is shown in Figure 9.

```
typedef struct  pbiovec {
        void    *data;
        IOFormat format;
} *pbiovec;

extern int
writev_IOfile (IOFile iofile, pbiovec vec, int count);
```

Figure 9: Prototypes for *writev_IOfile*().

4.5 Error Handling

Most of the routines described in the sections above can detect or return some kind of error condition. Some of them are verbose about it, printing error messages to `stderr`, but most return a particular value when they fail, generally `NULL` for the routines which return pointers or opaque data types, like *register_IOrecord_format*(), or 0 for routines like *read_IOfile*() which normally return an integer. Robust programs should always check the return values of the functions they call. To facilitate error handling, PBIO provides two additional functions, *IOhas_error*(*IOfile*) and *IOperror*(*IOfile*, *char**). *IOhas_error*() returns 0 if no error has occurred on the IOfile specified as its parameter and 1 if an error has occurred. *IOperror*() is similar to the UNIX *perror*() function. If an error has occurred on the IOfile specified as its parameter, it prints a textual message describing the error and includes the string specified as its second parameter.

5 Standard Tools

The meta-information contained in a PBIO data stream allows the construction of general tools to operate on PBIO files. Two such tools are provided with the PBIO library, IOdump and IOsort.

IOdump is a "cat" program for PBIO files. IOdump takes as arguments a set of options and a filename. By default IOdump prints an ascii representation of all data records and comments in the PBIO file. Dumping of record format information as well as header information in the file is enabled by specifying options of `+formats` and `+header` respectively. In general, printing for any record type can be turned on or off by specifying options of the form $\{+,-\}\{header, comments, formats, data\}$.

IOsort is a generalized sorting program for PBIO files. It takes three parameters, the name of the field to sort upon, the name of the input file and the name of the output file. The sort field can be of any PBIO atomic data type, but it must be the same basic type in all records. Any records in which the sort field does not appear will not appear in the output file.

6 PBIO With Other Networks

The basic PBIO routines above provide for using PBIO for I/O over file descriptors. This is sufficient for normal file and TCP/IP socket uses. But sometimes it is useful to use PBIO in other network circumstances

[3] UNIX typically restricts the number of independent memory areas that may be written with *writev*(). If the number and nature of the PBIO data to be written exceeds this limit, multiple calls will be performed.

as well. PBIO contains two types of support for transmission over networks without using a TCP/IP layer. The first type of support is designed for networks which still provide a TCP-like reliable, in-order, two-ended connection. The second provides a broader kind of support which makes no assumptions about the underlying network.

6.1 Customizing PBIO Low-Level I/O

To support binary I/O over networks providing a TCP-like interface, PBIO allows the substitution of user-supplied low-level I/O routines for those normally used to operate on the network interface. To support the full range of PBIO operations, the user must supply routines to read and write the network interface, a function to poll the network interface for new data and a routine to close the network interface. A typical call sequence to create a new PBIO file with a different low-level I/O interface follows:

```
{
    IOFile iofile = create_IOfile();  /* initialize IOfile structure */
    void *conn_info;
    /*
     * user code to create a network connection.  Upon creation,
     * place any information necessary to use the connection in
     * conn_info.  This pointer will be supplied to the read and
     * write routines.
     */
    set_conn_IOfile(iofile, conn_info);  /* assoc net info with iofile */

    set_interface_IOfile(iofile, new_write_func, new_read_func,
                         new_writev_func, new_readv_func, new_max_iov,
                         new_close_func, new_poll_func);

    open_created_IOfile(iofile, "w");  /* open file for writing */

    /* after this, operate on iofile with normal calls */
}
```

In the code above, the six functions **new_write_func, new_read_func, new_writev_func, new_readv_func, new_close_func** and **new_poll_func** are the new user-supplied network access routines. **new_write_func** and **new_read_func** are both of type **IOinterface_func** as defined in io_interface.h:

```
typedef int (*IOinterface_func)(void *conn, void *buffer, int length,
                                int *errno_p, char **result_p);
```

When PBIO calls these routines, the value used in the set_conn_IOfile() call will be supplied as the **conn** parameter. **buffer** and **length** specify the buffer to be read or written. Upon correct completion, the routines should return the number of bytes read or written. In the event of end of file, the function should return 0. In the event of other error, the **errno_p** or **result_p** parameters are used to specify the nature of the error. If the error is one which maps to a standard UNIX errno value, the integer pointed to by errno_p should be set to that value. If not, the **char*** pointed to by **result_p** should be set to a string describing the error. (This string will not be free()'d by PBIO. If it is not stored in static memory, the user routines should ensure that it is deallocated.)

The routines **new_readv_func** and **new_writev_func** are called by PBIO to read or write multiple buffers at a time. They are of type **IOinterface_funcv**:

```
typedef int (*IOinterface_funcv)(void *conn, struct iovec *iov, int icount,
                                 int *errno_p, char **result_p);
```

They are like the read and write functions except that instead of a single buffer and length, they take a vector of type **struct iovec** and a buffer count value. Type struct iovec is an address,length pair. If the

network interface imposes a maximum size on the number of buffers which can be efficiently read or written in one call, that value should be specified as the `max_iov` value in the `set_interface_IOfile()` call. If the network interface provides no direct multi-buffer read or write primitives, the readv and writev functions can be specified as NULL. In this case, PBIO will instead generate multiple calls to the single-buffer read and write routines.

The `new_close_func` and `new_poll_func` routines each take only the `void *conn` value as a parameter. The close function should shut down the network connection and free all resources associated with it. Poll should return zero if there is no data currently available for reading on the network interface. PBIO does not use this function except in the routine `poll_IOfile()`. Consequently, if the network interface does not provide any means of polling the network for data, this function can possibly be left NULL with no ill effects. If `poll_IOfile()` is called when the poll function is NULL, `poll_IOfile()` arbitrarily returns 0 (I.E. no data pending).

6.2 PBIO with Arbitrary Networks

There are many network transport mechanisms which are not TCP/IP-like in one way or another. Some relax reliability or ordering characteristics to achieve higher performance. Others vary topology to support such things as multiple receivers (broadcast and multicast) or multiple senders (with incoming messsages multiplexed over a single connection). Different applications may choose to use one of these network protocols for the actual transport of data but would still find it useful to use PBIO to handle the heterogeneity issues. Unfortunately, it is difficult for PBIO to deal directly with a non-TCP/IP-like network interface. In particular, the basic PBIO functions assume that the input appearing at the "read" port of an IOfile should be exactly what is written from the "write" side of a single IOfile. This is a natural assumption for TCP/IP like transports, but it breaks down if the underlying transport mechanism allows multiplexing of writers or readers or if it does not guarantee reliable, in-order delivery. PBIO could theoretically tolerate the loss or misordering of complete data records, but multiplexing of streams, delivery partial data records or any disruption affecting record format information would render some data uninterpretable.

In order to allow applications to gain the functional benefits of PBIO while using network transport mechanisms which PBIO can not directly support, we have defined an interface that allows separating the processes of data transport and format delivery. With this interface, instead of using *write_IOfile*() to transmit data, applications can *encode* the data in a form that can be interpreted by another PBIO library. The application can then transmitted this encoded buffer via any mechanism at its disposal. Once at its destination, the encoded buffer can then be passed to PBIO for decoding. Throughout this process, the transmission of format information from the encoder to the decoder is handled transparently by PBIO. Formats still need to be registered on the encoding side and conversions need to be set on the decoding side, but record format information travels from the encoder to the decoder via private reliable connections established by PBIO. Section 6.2.2 will discuss format communication in more detail.

The programs below show a simple use of these functions to transfer data. Here, the data is encoded in one program, "transmitted" by writing it to a file in one program and reading it in another,[4] and finally decoded in the "receiving" program.

[4] Of course, storing data in a file is the normal function of the {read,write}_IOfile interface. Files are used here only as an example transport mechanism.

```
void main()     /* sending program */
{
    IOContext src_context = create_IOcontext();
    IOFormat dyn_rec_ioformat;
    dyn_rec rec;
    int buf_size, fd, i;
    char *encoded_buffer;
    dyn_rec_ioformat = register_IOcontext_format("dynamic format",
                                                 dyn_field_list,
                                                 src_context);
    rec.string = "Hi Mom!";
    rec.icount = 5;
    rec.double_array = (double*) malloc(sizeof(double) * 5);
    for (i=0; i<5; i++)
        rec.double_array[i] = i*2.717;
    encoded_buffer = encode_IOcontext_buffer(src_context,
                        dyn_rec_ioformat, &rec, &buf_size);

    /* "transmit" encoded record over a file */
    fd = open("enc_file", O_WRONLY|O_CREAT|O_TRUNC, 0777);
    write(fd, encoded_buffer, buf_size);
}

void main()     /* receiving program */
{
    IOContext dest_context = create_IOcontext();
    IOFormat dyn_rec_ioformat;
    dyn_rec rec;
    int fd, i;
    char encoded_buffer[2048];  /* hopefully big enough */

    /* "receive" encoded record over a file */
    fd = open("enc_file", O_RDONLY, 0777);
    read(fd, encoded_buffer, sizeof(encoded_buffer));

    dyn_rec_ioformat = get_format_IOcontext(dest_context, encoded_buffer);
    set_conversion_IOcontext(dest_context, dyn_rec_ioformat,
                             dyn_field_list, sizeof(dyn_rec));
    decode_IOcontext(dest_context, encoded_buffer, &rec);
    printf("string is %s\n", rec.string);
    printf("icount is %d\n", rec.icount);
    for(i=0; i< rec.icount; i++)
        printf("element %d is %g\n", i, rec.double_array[i]);
}
```

The example programs above are quite simple, but they illustrate two basic characteristics of the IOContext interface:

- record formats and conversions are registered and set in an `IOContext` value, much as they are in `IOFile`'s in the connected interface. However, unlike the `IOFile`, the `IOContext` is not directly associated with a file, socket or other transmission or storage mechanism. It only serves as a placeholder to hold registration and conversion information.
- The example uses files as a "transport" mechanism, but PBIO isn't involved in the movement of the encoded data. Actual transmission from encoder to decoder could be via TCP, UDP or paper-tape clutched by a carrier pigeon. All that matters to PBIO is that the block of data presented to `decode_IOcontext()` must be one that resulted from a call to `encode_IOcontext()`.

The extreme simplicity of the examples allows the omission of details that would complicate the logic of more general programs. For example:

- The receiving program sets a conversion for the record without examining its format, implicitly assuming that it is a format that can be converted to the type `dyn_rec`. A complex pro-

gram supporting multiple record formats would presumably examine at least the name and perhaps the fields of the format before deciding which conversion to register for it. The function `has_conversion_IOformat(IOFormat ioformat)` returns true if a conversion has already been registered for a particular format, false otherwise.

- Additionally, to support record formats which might have nested subformats such as the record described in Figure 5, the use of an additional functions. In particular, `get_format_IOcontext()` returns only the top-level format of the record. But conversions must be set for any nested formats before a top-level conversion can be set. The PBIO routine `get_subformats_IOcontext()` (prototype below) returns a NULL-terminated list of IOFormat values corresponding to any nested formats contained within the record. The last entry in the list is the format of the record itself. The format list returned by this function should be free'd when no longer needed.

```
extern IOFormat *
get_subformats_IOcontext(IOContext iocontext, void *buffer);
```

- Because the examples just exchange a single record, they do not need to explicitly mark the beginning or end of the encoded data blocks. Because attempting to decode an incomplete or corrupted data block is likely to have unpredictable and deleterious consequences, applications should take steps to ensure that the data blocks they decode are always the complete and uncorrupted result of some encode operation.

- The examples each use a single IOContext and use it for a single operation. Individual IOcontext values can be used for both encoding and decoding and those operations will not affect each other. Many applications can get by with a single IOContext value per process, but more may be used to accommodate different threads of control or to ease management of the temporary memory provided by IOcontexts.

6.2.1 Memory Handling in the IOContext Interface

As with the IOFile interface, the memory use characteristics of the IOContext interface require documentation.

On the encoding side, the encoded data block is stored in temporary memory internal to PBIO and specific to the IOContext value used to encode the data. The memory will remain valid until the next encode or decode operation associated with that IOContext value. Currently, all data is copied into this temporary buffer so that it can be presented as a contiguous block. In the future PBIO may expose an `iovec`-style interface on the encoding side to avoid the copy operation.

On the receiving end, PBIO offers three different decode operations. The operations `decode_IOcontext()` and `decode_to_buffer_IOcontext()` are analogous to `read_IOfile()` and `read_to_buffer_IOfile()`. In both cases, the encoded data is preserved and the base record structure (excluding strings and dynamic arrays) in placed in memory provided by the user. In `decode_IOcontext()` (as in `read_IOfile()`), strings and dynamic arrays are copied into PBIO temporary memory associated with the decoding IOContext. The contents of that memory will remain valid until the next encode or decode operation performed in that context. However, in `decode_to_buffer_IOcontext()`, all data is placed in user-supplied memory. The routine:

```
extern int
this_IOrecord_length(IOContext context, char *src, int encoded_record_length);
```

returns the number of bytes of user memory required to hold the decoded record. This call requires the length of the encoded record block as a parameter and returns a conservative estimate of the number of bytes required to decode the record.

The third decode option offered by PBIO is `decode_in_place_IOcontext()`. The first two calls, `decode_IOcontext()` and `decode_to_buffer_IOcontext`, always leave their output in a different location than where the encoded buffer resides. This requires even data fields that need no significant modification to be moved to a different location and may not the most efficient way to decode the record. This is particularly true for transfers between homogenous targets. In contrast, `decode_in_place_IOcontext()` attempts to decode the message and leave it where it is in memory. Where this is possible it is at least as efficient as `decode_IOcontext()` and perhaps much more efficient because it need not involve any significant data movement. However, sometimes it isn't possible to decode the buffer in place. Sometimes the fields in the current and the desired structure overlap in ways that make it difficult to rearrange them in place. When this kind of conflict happens, `decode_in_place_IOcontext` must copy the encoded buffer into a temporary location (managed by PBIO), and then do a normal `decode_IOcontext()` with the source being the temporary buffer and the destination being where the data used to sit.

Despite differences in performance, each call will always maintain the appropriate semantics. That is, `decode_in_place_IOcontext()` will always leave the result in the same memory as the encoded buffer and `decode_IOcontext()` will always leave the result in the specified destination buffer. Some programs may choose to ignore the performance differences and use one call or the other depending upon which of those semantics best fits the memory use requirements of the application. However, smarter applications can achieve the best of both worlds by making use of the `decode_in_place_possible()` routine which separates the two possible situations. The following example demonstrates its use.

```
/* buffer msg contains encoded message */
IOFormat format = get_format_IOcontext(context, msg);

if (decode_in_place_possible(format)) {
    char *msg_start;
    decode_in_place_IOcontext(context, msg, &msg_start);
    /*
     * note that with an in-place decode, byte 0 of the resulting
     * decoded structure is probably not at byte 0 of the encoded
     * buffer.  Therefore, msg_start is probably a few bytes into
     * the buffer msg.  The precise relationship between msg and
     * msg_start is a PBIO implementation detail.
     */
    process_message(msg_start);
} else {
    int length = this_IOrecord_length(context, msg, incoming_size);
    char *new_msg = malloc(length);
    decode_to_buffer_IOcontext(context, msg, new_msg);
    process_message(new_msg);
    free(new_msg);
}
```

6.2.2 Communication of Format Information

While the actual communication of record format information from the encoding context to the decoding context is handled entirely within PBIO, the manner in which this communication is accomplished may influence the behavior of the application in some ways.

Eventually, we envision a variety of mechanisms through which record format information from IOContexts can be stored and exchanged. However, at the moment there is only one supported mechanism with a limited amount of customizability. When a `register_IOcontext_format()` is done, PBIO synchronously and reliably transmits record format information to a third party, a *format server*, running on a known host and listening at a known port. Then, PBIO waits for the format server to return to it a 64-bit format ID. The `register_IOcontext_format()` operation returns only when the format ID has been received at the encoding side and stored with the IOContext value. When data for that format is encoded, PBIO appends the format ID to the data to identify the format. On the receiving side, `get_format_IOcontext()` extracts

the 64-bit format ID from the data and looks in the decoding IOContext to see if that particular format ID has been seen before. If not, PBIO synchronously queries the format server and retrieves the format of record. Thereafter, the format information is associated with the format ID in the IOContext value and no further contact with the format server is necessary to process records of this type.[5]

The use of a centralized format server as a data repository has several side effects. As a centralized resource, the format server is a possible bottleneck in a large system. To mitigate this effect, the format server consolidates identical record formats registered by hosts of the same architecture and assigns them all the same format ID. This means that a host receiving the same type of information from N machines of M different types will only query the format server M times. Another side effect is that the format server is subject to periodic information loss due to power outages, etc. This possibility limits the length of time a format ID will remain associated with a particular set of format information, and therefore limits the life of encoded data. Since these events are rare and the IOFile() interface is available for long-term data storage, we have no current plans to address this limitation in the IOContext interface. At this time the location (host and IP port) of the format server is a compile-time parameter to PBIO.

7 Conclusion

This paper has presented an overview of the PBIO library for self-describing binary data streams. The library provides for machine independent representation of binary data and meta-data in both files and streams.

References

[1] Ruth A. Aydt. *The Pablo Self-Defining Data Format.* Picasso Research Group, Department of Computer Science, University of Illinos at Urbana-Champaign, April 1994.

[2] NCSA. *NCSA HDF.* University of Illinois at Urbana-Champaign, National Center for Supercomputing Applications, February 1989.

[3] R. K. Rew. *netCDF User's Guide.* Unidata Program Center, University Corporation for Atmospheric Research, April 1989.

[5] However, as this association is maintained in the IOContext value, the use of multiple IOContexts may require multiple contacts to the format server.