

# Portable Self-Describing Binary Data Streams

**Greg Eisenhauer**  
eisen@cc.gatech.edu

College of Computing  
Georgia Institute of Technology  
Atlanta, Georgia 30332

## 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 P BIO Library

The basic approach of the Portable Binary I/O library is relatively simple. P BIO 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 P BIO read routine performs the appropriate translations.

### 3.1 Supported Data Types

The P BIO 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 P BIO 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 P BIO scheme, the field list which governs the writing *IOFormat* becomes the *file record format* in the written file.

In the case of reading a P BIO 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 P BIO 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.<sup>1</sup> The source program supplies the complete specification of the data being written. The destination program specifies the fields 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 P BIO routines perform

---

<sup>1</sup>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", O_WRONLY|O_CREAT|O_TRUNC);
    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.

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 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

```

void
main(argc, argv)
int argc;
char **argv;
{
    IOFile iofile = open_IOfile("test_output", O_RDONLY);
    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.

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 P BIO library. It is possible to use either approach with the P BIO 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 P BIO 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 P BIO 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

P BIO 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 two dimensional array floating point numbers. At present, P BIO supports only fixed array sizes of one or two dimensions. The field size specified should be the size of the entire array, not just the size of each element. When reading a record containing array fields, the

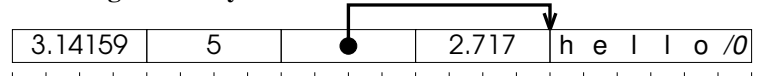
**Program Structure:**

```
struct {  
    float x;  
    int y;  
    char *str;  
    float z; }  
}
```

**Program Structure Length: 16 bytes**

**Incoming File Record: { 3.14159, 5, "hello", 2.717 }**

**Incoming buffer layout:**



**Required buffer length: 22 bytes**

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

dimensionality of each field must match that which was used when the record was written, though the size of the elements may differ.

Additionally, 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 4 could be written by registering formats defined by these field lists:

```
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},
};
```

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

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

Figure 4: A nested record format

### 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 5 and 6 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 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 7 shows the body of a reader program that is capable of handling new formats at any time. Its

```

first_rec_ioformat = register_I0record_format("first format", field_list, iofile);
vec_ioformat = register_I0record_format("R3vector", R3field_list, iofile);
particle_ioformat = register_I0record_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_I0file(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_I0file(iofile, particle_ioformat, &p)) {
            printf("write failed\n");
        }
    }
}
}

```

Figure 5: The body of a more complex writer program.

```

I0File iofile = open_I0file("test_output", O_RDONLY);
I0Format first_format, particle_format, next_format;

set_I0conversion(iofile, "first format", field_list, sizeof(first_rec));
set_I0conversion(iofile, "R3vector", R3field_list, sizeof(R3vector));
set_I0conversion(iofile, "particle", particle_field_list, sizeof(particle));

first_format = get_I0format_by_name(iofile, "first format");
particle_format = get_I0format_by_name(iofile, "particle");

next_format = next_I0record_format(iofile);
while(next_format != NULL) {
    if (next_format == first_format) {
        first_rec rec1;
        read_I0file(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_I0file(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_I0record_format(iofile);
}
}

```

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

```

IOFile iofile = open_IOfile("test_output", O_RDONLY);
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 7: A reader program for dynamic format registration.

organization is somewhat different from the previous reader program of Figure 6. 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 P BIO and the `read_comment_IOfile()` call returns a pointer to this buffer. The buffer contents are only guaranteed valid until the next P BIO 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

P BIO 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_IRecord_count()` returns the number of array records pending.<sup>2</sup>

The other facility for bulk record handling is the routine `writenv_IOfile()`. `writenv_IOfile()` is similar to the `writenv()` system call. Instead of taking single data buffer (of a particular format) to write, `writenv_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 `writenv()` system call.<sup>3</sup> `writenv_IOfile()` imposes no restrictions on the nature of the fields in the records to be written. Unfortunately, the nature of the P BIO protocol allows no corresponding read call. Records written a single `writenv_IOfile()` must be read with multiple `read_IOfile()` calls. The prototype of `writenv_IOfile()` is shown in Figure 8.

## 5 Standard Tools

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

IODump is a “cat” program for P BIO 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 P BIO file. Dumping of record format information as well as header information in the file is enabled by specifying options of

---

<sup>2</sup> “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.

<sup>3</sup> UNIX typically restricts the number of independent memory areas that may be written with `writenv()`. If the number and nature of the P BIO data to be written exceeds this limit, multiple calls will be performed.

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

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

Figure 8: Prototypes for *writev\_IOfile()*.

**+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.

## 6 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 Illinois 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.