`eventio` – a machine-independent hierarchical data format and its programming interface[*]

K. Bernlöhr

July 17, 2014

Contents

[*]Originally developed for the Cosmic Ray Tracking (CRT) project in 1992 to 1996. Used also for the HEGRA system of Cherenkov telescopes from 1995 to 2002 and for Monte-Carlo simulations of many different experiments. Source code available under open-source licences.

1 Design criteria

Because of different architectures of the computers with which experimental data should be read-out (at that time Motorola 680x0 under OS-9) and those used for data analysis (like DEC Stations under Ultrix, VAX under VMS, IBM RS/6000 under AIX, and other systems introduced later), the data format had to be *machine-independent*. Data written on any supported computer type should be readable for all others.

Input and output should be, nevertheless, very efficient. To avoid being limited to a particular processor type, the data format should support several internal formats and convert as appropriate. Since internal formats of all CPUs involved only differ in the byte order (0-1-2-3 or 3-2-1-0), except for the floating-point representation of the VAX, it is sufficient to record the byte order used together with the data. Only for the VAX, an additional conversion of floating point numbers was required.

If source and target computers have different byte orders, a conversion is required on one of them. Since the `eventio` data format allows to use either byte ordering, this conversion can be done on the machine where CPU time is least critical. If off-line processing, for example, is done on just one type of architecture, it would be most efficient recording the data in the native byte order of that architecture. Other types of machines would – with a small loss of efficiency – still be able to read the data.

Errors while writing, reading or transferring the data should not result in corruption of all data following. Therefore, every major data block is preceded by a 4 bytes long synchronization marker. This marker is recognized in either byte order and serves, at the same time, to indicate the byte order used for writing.

For flexibility in writing different data types into one data stream and to be able to filter a selection of data types from a composite data stream, all data blocks have type and identification (ID) fields. The type represents the scheme how a block is built and should corresponds to a single function needed to read the block. The ID would typically identify the origin of the data (e.g. the detector) but other uses are not a problem. In order to be prepared for later changes in the definition of any type of data block, the data blocks also have a version field. In order to be able to skip data blocks which are not of interest or for which no decoding function is known to the program, a length field is included as well.

For flexibility in building up blocks of any type, a hierarchical structure is used. Each major block of data can contain a tree of sub-blocks as appropriate, with lowest-level blocks only containing *elementary data types*. In view of this flexibility it is more appropriate to talk of data objects or items than of blocks. The term ‘block’ implies a too fixed and too simple structure for describing the `eventio` data structure. Reading or writing one type of objects also can usually be represented by a method in object-oriented programming (for example in C++) although the basic programming interface is entirely written in portable C. As long as any of these objects consists of sub-objects only, but does not contain elementary data, the structure of it can be determined without knowing anything about the objects or having a decoding function available for it. Sub-objects can be searched for or skipped at any level, and they can even be deleted without from the tree without disturbing the consistency of the major objects structure. Only the major or top level object (in the tree picture: the root) is prefixed by the synchronization marker. In principle, the byte ordering can be changed from one major object to the next one.

For efficiency reasons every object is filled to the next 4-bytes boundary when ‘writing’ on (actually: encoding it into an internal I/O buffer). This way, every object should usually start a 4-bytes address, to reduce accessing addresses which cross word limits and have slower memory access on many CPU types than have word-aligned accesses. To take advantage of that requires, of course, that elementary data types within the low-level objects are also aligned. If an object is made up of a mixture of elementary data types and sub-objects, this efficient alignment cannot be guaranteed for the sub-objects. Except for efficiency, however, no side-effects should happen (for example on machines which cannot access integer of floating point numbers on odd memory addresses). The `eventio` software does take care of that.

2 Data types and data structures

2.1 Elementary data types

As elementary data types most of those also known in programming languages as C or Fortran are implemented:

Type	Length (Bytes)	Description
Byte	1	character or very small number
Short	2	short integer (with or without sign)
Long	4	longer integer (usually an `'int'`, with or without sign)
Int16	2	16-bit signed integer = Short
UInt16	2	16-bit unsigned integer
Int32	4	32-bit signed integer = Long
UInt32	4	32-bit unsigned integer
Int64	8	64-bit signed integer
UInt64	8	64-bit unsigned integer = Long64
Count	1 to 9	Natural number (unsigned)
SCount	1 to 9	Signed natural number
SFloat	2	Float16, 16-bit floating point number (as in OpenGL)
Real	4	32-bit floating point number in IEEE format
Double	8	64-bit floating point number in IEEE format
String	2+Length	string of characters or bytes prefixed with a length up to 32767.
LongString	4+Length	string of characters or bytes prefixed with a length up to 2147483647.
VarString	var.+Length	string of characters prefixed with length.

The 64-bit (8-byte) integers (*Long64*) are only possible on machines with support for such data types. That includes systems where 64-bit integers are only implemented through the compiler (e.g. as `long long`). Beware that the data may not be readable on machines without 64-bit integer support, unless special provisions are made, like reading the data into two 32-bit integers. With the Count and SCount type elements, it is also possible to write values on a 64-bit capable system that cannot be properly read on a system without 64-bit integer support.

Note that the *Long* type does match an `int` rather than a `long` C variable type on 64-bit machines. The elementary data types are normally used without any declaration included in the data. The decoding software, therefore, has to know how to deal with the data.

The synchronization marker is written and read as a *Long* type. If read as 0x378A1FD4 instead 0xD41F8A37, the byte order of all elementary data types has to be reversed, except for *Strings* where only the byte order of the leading length value needs to be reversed. Strictly speaking, all integer types should be distinguished between signed and unsigned types. As long as the data is returned to variables of the same type, this should not pose a problem. All machines supported use (except for byte order) the same representation of negative numbers. On 32-bit machines, sign propagation of *Short* types to `int` or `long` variables has to be taken care of and, on 64-bit machines, also of *Long* types to `long` variables.

2.2 Object structure

The elementary data types are combined to objects which, in turn, can be part of a hierarchical structure. Input and output, definition of the byte order as well as synchronization in case of data corruption are all done for the major (top-level) object only. The structure of the top-level object and its sub-objects only differs by the fact that the top-level objects has a 4-bytes synchronization marker prefix one or the other byte order. This marker consists of the integer 0xD41F8A37 (in C notation, i.e. 3558836791 as an `unsigned long` or -736130505 as a `signed long`. Apart from that, every object has a header of three 4-byte words, representing the *type/version*, *identification*, and *length* fields. The body of an object is the *data field*.

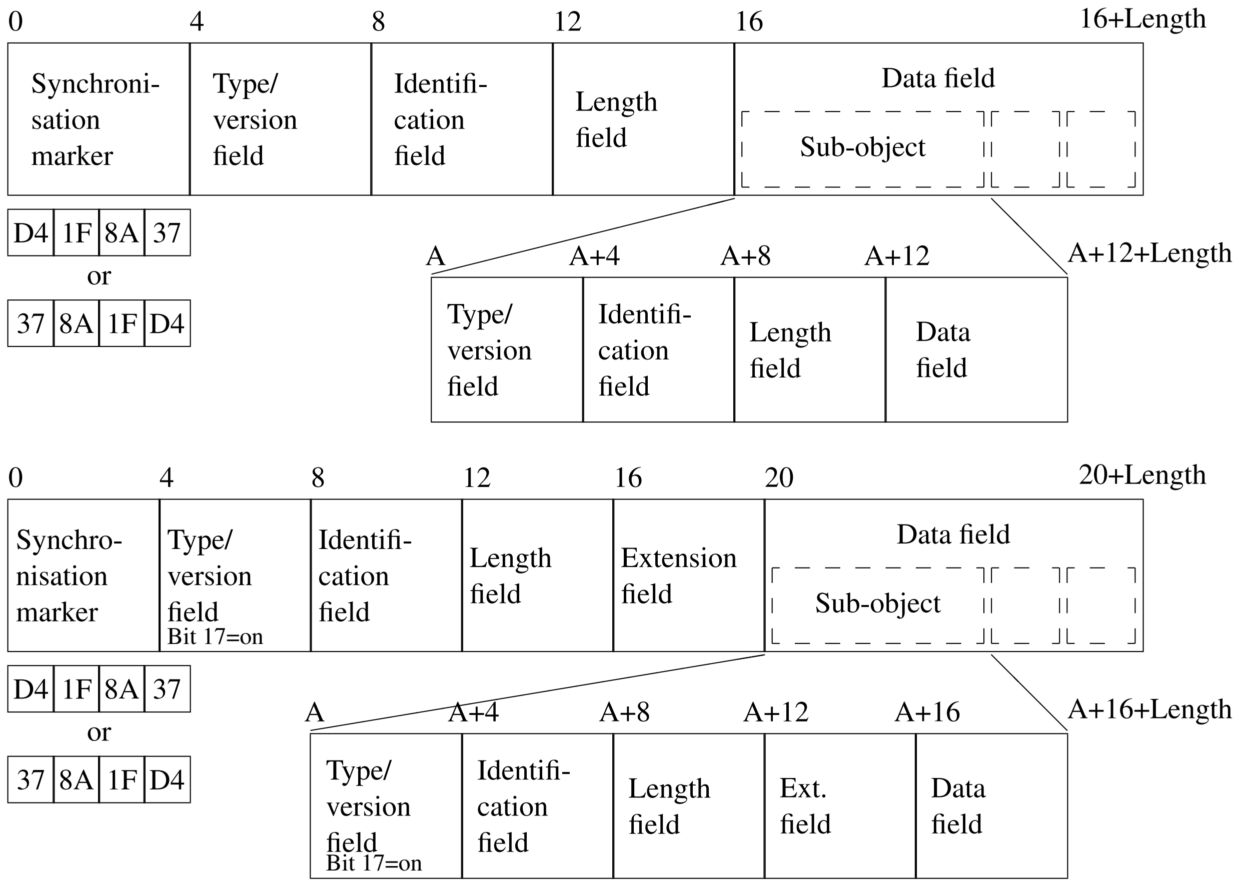

Figure 1: Schematic structure of a top-level object and a sub-object (top: without any extension, bottom: top-level object and sub-object with extension active).

The type/version field combines a type number in bits 0 to 15 (with values between 0 and 65535) and a version number in bits 20 to 31 (values between 0 and 4095). Bit 16 is defined as 'User Bit' and no interpretation is defined. If Bit 17 is set, it indicates that the extended format is in use - which implies an addition 4-byte header field, the extension field. Bits 18 and 19 are reserved for future enhancements and have to be 0 as long as this definition is not changed.

The ID field should identify from where the data come, which event they represent, or something of that kind. Thus, the ID field has an object-specific meaning. An ID of -1 should be interpreted as 'no ID given.'

The length field contains in bits 0 to 29 the length of the data field (in bytes), including 0 to 3 fill bytes at the end to fill up to a length that is divisible by 4. As such, the length should be in the range 0 to 1073741823 – and divisible by 4. Bit 30 is set if the object contains only sub-objects but no elementary data types, and therefore its contents can be listed without knowing anything about specific object types. Bit 31 is reserved and has to be 0.

If bit 17 of the type/version field is set, the extension field follows, with only bits 0-11 defined so far, extending the 30 bits from the length field by another 12 bits, thus supporting data blocks of up to 4 TiB length. All other bits are reserved and must be 0 for now.

A data field can contain sub-objects or elementary data types. If containing elementary data types (or a combination of those and of sub-objects), it can only be decoded by an object-type-specific function. If containing only sub-objects then bit 30 of the length field will be set and its contents can be listed and sub-objects can be searched for, without type-specific knowledge.

3 Implementation and programming in C

3.1 Overview

With the present implementation for the `eventio` data format the top-level objects are always assembled in memory before any writing is done and they are read completely into memory before decoding is started. There is only one top-level object in a single I/O buffer. The available functions can be divided into five groups:

1. memory management for I/O buffers,
2. I/O, search for and skipping of top-level objects,
3. management of objects in an I/O buffer,
4. writing/reading of elementary data types into/from an I/O buffer, and
5. writing/reading of complete objects/object hierarchies (type specific).

For the management of I/O buffers and of objects two types of structures are used, of which in the following only those elements are shown which are intended to be used directly by application-specific functions:

```
struct _struct_IO_BUFFER
{
   ...
   int input_fileno;
   int output_fileno;
   FILE *input_file;
   FILE *output_file;
   int (*user_function)(unsigned char *, long, int);
   ...
};
typedef struct _struct_IO_BUFFER IO_BUFFER;

struct _struct_IO_ITEM_HEADER
{
   ...
   unsigned long type;
   unsigned version;
   long ident;
   ...
};
typedef struct _struct_IO_ITEM_HEADER IO_ITEM_HEADER;
```

In the `IO_BUFFER` structure the elements `input_fileno` and `output_fileno` are file handle numbers as obtained by opening a input or output file with `open()` or by applying `fileno()` to an available `FILE` structure. They are 0 for standard input or 1 for standard output. When an `IO_BUFFER` structure is allocated and initialized, these variables are set to -1, indicating that no file handles have been assigned. If the `IO_BUFFER` is only used for input, only `input_fileno` needs to be set, and correspondingly only `output_fileno` for output.

Instead of I/O via the basic I/O functions `open`, `read`, `write`, and `close`, the 'stream' I/O functions `fopen` (or `popen`), `fread`, `fwrite`, and `fclose` (or `pclose`) can be used as well. For this purpose the elements `input_file` and/or `output_file` should be set with a pointer to a corresponding `FILE` structure, instead of setting `input_fileno` or `output_fileno`. Data with small blocks sizes gains quite a lot in performance from the internal buffering by the stream I/O functions.

If I/O should use some other means rather than files or pipes then, alternatively, the function pointer `user_function` can be set to user-defined function of the corresponding type. Then this function has to take care of the 'real' I/O.

In each `IO_ITEM_HEADER` structure the type number (`type`), the version number (`version`) and the ID number (`ident`) should be set before starting with a new data object. Before searching or reading, the type field can be given (any mismatch resulting in an error) while other fields will be set to values from the data encountered.

3.2 Management of I/O buffers

`IO_BUFFER *allocate_io_buffer(size_t length);`
This function allocates and initializes an I/O buffer which will have an initial buffer size of `length` bytes but can be dynamically extended when needed. After that the file handles or file pointers or function pointer can be assigned for the I/O buffer.

Example: `iobuf = allocate_io_buffer(20000);`

The actual size of the buffer can increase dynamically, up to a user-defined maximum size, for example

```
      iobuf->max_length = 10000000;
```

`void free_io_buffer(IO_BUFFER *iobuf);`
An I/O buffer should be deleted. Note that the corresponding files used for I/O will not be closed by this function and a user-defined I/O handler function would not be notified.

Example: `free_io_buffer(iobuf);`

3.3 Low-level input and output functions

`int write_io_block (IO_BUFFER *iobuf);`
After completing a top-level object and all the data contained in it, including any sub-objects the contents of the buffer will be written into the file referred to by `iobuf->output_fileno` or `iobuf->output_file` or, if neither of those is set but `iobuf->user_function` is, the actual buffer contents and length will be passed to the used function (and a `code` value of 1).

Example:
```
if ( (rc = write_io_block(iobuf)) != 0 )
        return rc;
```

`int find_io_block (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header);`
In the file referred to by `iobuf->input_fileno` or `iobuf->input_file` the next synchronization marker will be searched for. Usually this will be just the next four bytes in the input file. Then the remaining head of the top-level object will be read into the I/O buffer and the described elements of `item_header` set accordingly. If `iobuf->user_function` is used, the user function has to take care of getting to the next top-level object and storing the item header in the provided buffer:

```
int (*user_function) (unsigned char *buffer, long bytes, int code);
```

with a `code` value of 2 (see below).

`int read_io_block (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header);`
Used to read the data field of the top-level object found with the preceding `find_io_block` call (user function would be called with `code` 3). After that, the data is available for decoding.

`int skip_io_block (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header);`
Used to skip over the data field of the top-level object found with the preceding `find_io_block` call. This is usually done when not interested in a particular type of data object. A user function would be called with `code` 4.

Example:

```
IO_ITEM_HEADER item_header;
...
if ( find_io_block(iobuf,&item_header) != 0 )
   printf("Error!\n");
else if ( item_header.type == 2540 )
{
   if ( read_io_block(iobuf,&item_header) != 0 )
      printf("Also an error!\n");
   ...
}
else
   (void) skip_io_block(iobuf,&item_header);
```

`int list_io_blocks (IO_BUFFER *iobuf);`
This is already a simple application based on the low-level functions. It produces a listing of all top-level objects found in the input file, with output going to standard output. A complete, although simple application program would be the following:

```
#include "io_basic.h"
int main()
{
   IO_BUFFER *iobuf;
   if ( (iobuf = allocate_io_buffer(1000)) ==
        (IO_BUFFER *) NULL )
      exit(1);
   iobuf->input_fileno = 0;
   (void) list_io_blocks(iobuf);
   exit(0);
   return 0;
}
```

If I/O is not file-oriented or at least not to be done with the standard functions then, after I/O buffer allocation, the element `iobuf->user_function` can be set to the address of a user-defined function. The elements for input or output file handles or structures should not be set in that case. The user-defined function must follow this prototype:
`int function (unsigned char *buffer, long bytes, int code);`
Among the parameters `code` defines what should be done:

`code=1`**:** Write `bytes` bytes starting at address `buffer`. This includes the header and the data.

`code=2`**:** Find the next top-level object and read its header (16 bytes) into the buffer pointer to by `buffer`. If the header indicates that the extension field is used, another four bytes will be requested from the user function with `code=3`, before getting or skipping the data field.

`code=3`**:** Read the data field of the top-level object which was found before. Here `bytes` is the known length of the data field and `buffer` is the address to which the data field should be copied (which is 16 bytes after the start of the I/O buffer or more if the extension field is used). The buffer is guaranteed to be large enough to hold the data.

`code=4`**:** Skip a data field of length `bytes` for the top-level object found before. `buffer` must not be used here, not even for temporary purposes, since for skipping of data objects no memory will be allocated.

The user-defined function must return 0 after proper execution. In case of errors (except end-of-file while reading) -1 should be returned and in case of end-of-file (for `codes` 2–4) a value of -2.

3.4 Management of data objects

For proper management of data objects, the functions `put_item_begin` and `put_item_end` are needed before and after putting the data into the I/O buffer. These functions are needed as a matching pair with respect to each object. Correspondingly, before reading (decoding) from the buffer, the function `get_item_begin` is needed as well as `get_item_end` afterwards. Additional functions serve for searching, deleting, backspacing, and listings. All functions return 0 in case of proper operation and negative numbers in case of errors.

```
int put_item_begin (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header);
int put_item_end   (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header);
int get_item_begin (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header);
int get_item_end   (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header);
int unget_item     (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header);
int rewind_item    (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header);
int remove_item    (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header);
int search_sub_item(IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header,
                    IO_ITEM_HEADER *sub_item_header);
int list_sub_items (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header,
                    int maxlevel);
```

Examples of functions used to write one object and to read it are included later in this paper.

After a successful call to `get_item_begin` or `search_sub_item`, the data pointer is at the beginning of the data field of the corresponding object or sub-object. Its data can now be decoded (read) or the object can be skipped. Usually, for every call to `put_item_begin` or `get_item_begin` there should be exactly one corresponding call to `put_item_end`or `get_item_end`. If a higher-level object is finished, however, all of its descendents will be closed too – although at the expense of a warning. The `put_item_end` call for the top-level object will start the low-level or real I/O part, depending on the settings for the object (typically writing to file).

If `get_item_begin` fails, the data pointer remains at the previous place. On the other hand, if `search_sub_item` fails the data pointer is at the end of the object in which a certain type of sub-object was searched for. The data pointer can be set back to the beginning of the data field with `rewind_item`. To get back even before the head of the object, use `unget_item`. To remove an object call `remove_item` after a successful `get_item_begin` or `search_sub_item`. No `get_item_end` is needed after removing. Function `list_sub_items` works similar to sub-objects as `list_io_blocks` to top-level objects, listing all sub-objects of a given object. If the argument `maxlevel` is larger than zero, only sub-objects up to the given nesting level will be mentioned. After `list_sub_items`, the data pointer is at the end of the corresponding object.

3.5 Use of elementary data types

For each of the elementary data types there is at least one function to put a single element into the I/O buffer and at least one function to get a single element from the buffer. In either case the data pointer is incremented correspondingly. Byte operations are implemented as preprocessor macros – just in case you know what kinds of restrictions this imposes. Except for Strings, there are also functions writing vectors of many elements, reducing some management overhead. Since the vector functions have no automatic conversion of arguments or return values, you should check that vector of the proper type and size are passed (important with pre-ANSI C compilers). This includes functions `put_vector_of_short` and `put_vector_of_int` and their inverse where you should know that only integers between -32768 and 32767 or between 0 and 65535 can be represented in the data (depending if the data is interpreted as signed or unsigned) but the elements of the vectors passed to the functions may have different lengths on different system (2 or 4 bytes long). Similarly, on 64-bit machines the `long` type holds more bits of data than the *Long* data type and only the 32 lower bits will be written. But as long as the number fits into the corresponding data type, there is nothing to worry.

Where data needs to be stored internally in variables of a specific length, you should use the ANSI C99 data types `int8_t`, `uint8_t`, ..., `int64_t`, `uint64_t` which, after including io_basic.h, should be available also for older compilers. Note that the functions involving 64-bit integers are not available on all machines.

Elementary functions:

```
int  put_byte (unsigned char c, IO_BUFFER *iobuf);
int  get_byte (IO_BUFFER *iobuf);
void put_count (uintmax_t i, IO_BUFFER *iobuf);
uintmax_t get_count (IO_BUFFER *iobuf);
void put_scount (intmax_t i, IO_BUFFER *iobuf);
intmax_t get_scount (IO_BUFFER *iobuf);
void put_short (int i, IO_BUFFER *iobuf);
int  get_short (IO_BUFFER *iobuf);
void put_int32(int32_t num, IO_BUFFER *iobuf)
int32_t get_int32(IO_BUFFER *iobuf)
void put_uint32(uint32_t num, IO_BUFFER *iobuf)
uint32_t get_uint32(IO_BUFFER *iobuf)
void put_long (long l, IO_BUFFER *iobuf);
long get_long (IO_BUFFER *iobuf);
void put_string (char *s, IO_BUFFER *iobuf);
int  get_string (IO_BUFFER *iobuf);
int put_long_string (char *s, IO_BUFFER *iobuf)
int get_long_string (char *s, int nmax, IO_BUFFER *iobuf)
int put_var_string (char *s, IO_BUFFER *iobuf)
int get_var_string (char *s, int nmax, IO_BUFFER *iobuf)
void put_real (double d, IO_BUFFER *iobuf);
double get_real (IO_BUFFER *iobuf);
void put_double (double dnum, IO_BUFFER *iobuf)
double get_double (IO_BUFFER *iobuf)
```

Vector functions:

```
void put_vector_of_byte (unsigned char *cv, int num, IO_BUFFER *iobuf);
void get_vector_of_byte (unsigned char *cv, int num, IO_BUFFER *iobuf);
void put_vector_of_uint8 (uint8_t *cv, int num, IO_BUFFER *iobuf);
void get_vector_of_uint8 (uint8_t *cv, int num, IO_BUFFER *iobuf);
void put_vector_of_short (short *sv, int num, IO_BUFFER *iobuf);
void get_vector_of_short (short *sv, int num, IO_BUFFER *iobuf);
void put_vector_of_int (int *iv, int num, IO_BUFFER *iobuf);
void get_vector_of_int (int *iv, int num, IO_BUFFER *iobuf);
void put_vector_of_int16 (int16_t *sv, int num, IO_BUFFER *iobuf);
void get_vector_of_int16 (int16_t *sv, int num, IO_BUFFER *iobuf);
void put_vector_of_uint16 (uint16_t *uval, int num, IO_BUFFER *iobuf)
void get_vector_of_uint16 (uint16_t *uval, int num, IO_BUFFER *iobuf)
void put_vector_of_int32 (int32_t *vec, int num, IO_BUFFER *iobuf)
void get_vector_of_int32 (int32_t *vec, int num, IO_BUFFER *iobuf)
void put_vector_of_uint32 (uint32_t *vec, int num, IO_BUFFER *iobuf)
void get_vector_of_uint32 (uint32_t *vec, int num, IO_BUFFER *iobuf)
void put_vector_of_int64 (int64_t *ival, int num, IO_BUFFER *iobuf)
void get_vector_of_int64 (int64_t *ival, int num, IO_BUFFER *iobuf)
void put_vector_of_uint64 (uint64_t *uval, int num, IO_BUFFER *iobuf)
void get_vector_of_uint64 (uint64_t *uval, int num, IO_BUFFER *iobuf)
void put_vector_of_long (long *lv, int num, IO_BUFFER *iobuf);
void get_vector_of_long (long *lv, int num, IO_BUFFER *iobuf);
void put_vector_of_real (double *dv, int num, IO_BUFFER *iobuf);
void get_vector_of_real (double *dv, int num, IO_BUFFER *iobuf);
void put_vector_of_float (float *fvec, int num, IO_BUFFER *iobuf)
void get_vector_of_float (float *fvec, int num, IO_BUFFER *iobuf)
void put_vector_of_double (double *dvec, int num, IO_BUFFER *iobuf)
void get_vector_of_double (double *dvec, int num, IO_BUFFER *iobuf)
```

3.6 Object-type specific functions

Two object-type specific functions are needed, one for 'reading' (i.e. decoding from the I/O buffer) and one for 'writing' (putting things into the I/O buffer). These functions should check for consistency of their arguments in the first place and pass any error conditions, e.g. concerning data object management or buffer management or real I/O, up to the calling function. The calling functions, in turn, should handle error conditions happening while processing sub-objects. A simple pair of functions for writing and reading a set of ADC pedestals follows. For more complex functions, the arguments passed usually include a pointer to an internal data structure (plus perhaps the number of elements to be written). The pointer to the I/O buffer should always be passed to the function.

```
#include "initial.h"      /* This file includes others as required. */
#include "eventio.h"      /* This file includes others as required. */

/* ---------------------- write_offsets ---------------------- */
/**
 *  @short Reads (previously measured) ADC offset values.
 *
 *  The destination ('offsets' parameter) will usually be the
 *  global 'cmpoffset' data.
 *
 *  @param offsets    The destination of the data being read.
 *  @param nchannels  The expected no. of channels, must match
 *                    the actual no. specified in the input.
 *  @param iobuf      The input data iobuf descriptor.
 *
 *  @return  0 (O.k.),  -1 (error),  or  -2 (e.o.f.)
 *
 */

int write_offsets (long *offsets, int nchannels, IO_BUFFER *iobuf)
{
   IO_ITEM_HEADER item_header;

   if ( iobuf == (IO_BUFFER *) NULL )
      return -1;
   if (offsets == (long *) NULL )
   {
      Warning("Attempt to save invalid offset vector");
      return -1;
   }

   item_header.type = 21;              /* Offset data is type 21 */
   item_header.version = 1;            /* Version 1 (preliminary) */
   item_header.ident = get_detector_number();
   put_item_begin(iobuf,&item_header);

   /* Save the time when the data is written although it would */
   /* be better to save the time when the offsets were measured. */
   put_long(time((time_t *) NULL),iobuf);
   put_short(nchannels,iobuf);         /* No. of FADC channels */
   put_vector_of_long(offsets,nchannels,iobuf);

   return(put_item_end(iobuf,&item_header));
}

/* ----------------------- read_offsets ------------------------ */
/**
 *  @short Reads (previously measured) ADC offset values.
 *
 *  The destination ('offsets' parameter) will usually be the
 *  global 'cmpoffset' data.
 *
 *  @param offsets    The destination of the data being read.
 *  @param nchannels  The expected no. of channels, must match
 *                    the actual no. specified in the input.
 *  @param iobuf      The input data iobuf descriptor.
 *
 *  @return  0 (O.k.),  -1 (error),  or  -2 (e.o.f.)
 *
 */

int read_offsets (long *offsets, int nchannels, IO_BUFFER *iobuf)
{
   IO_ITEM_HEADER item_header;
   int rc;

   if ( offsets == (long *) NULL )
   {
      Warning("Invalid call to read_offsets()");
      return -1;
   }

   if ( iobuf == (IO_BUFFER *) NULL )
      return -1;
   item_header.type = 21;
   if ( (rc = get_item_begin(iobuf,&item_header)) != 0 )
      return(rc);

   if ( item_header.version != 1 )
   {
      Warning("Wrong version no. of offset data to be read");
      return -1;
   }

   (void) get_long(iobuf);    /* Time is ignored, so far. */

   if ( nchannels != get_short(iobuf) )
   {
      Warning("Wrong no. of channels for reading offsets");
      return -1;
   }

   get_vector_of_long(offsets,nchannels,iobuf);

   return(get_item_end(iobuf,&item_header));
}
```

The comments at the beginning of the functions with special markups are for automatic generation of documentation with the `doxygen` program.

4 Implementation and programming in C++

4.1 Introduction

In addition to the C language basic implementation and API, there is an additional application programming interface in C++ which takes care of hiding many of those things the C programmer has to write explicitly. The C++ API also provides a much cleaner and simpler – even though richer and more powerfull – collection of methods to get data from or put data to the internal I/O buffer. Among the added benefits are support for standard library vectors, valarrays and strings.

You should be aware that you may have to link with a different library for the C++ API than for the C API.

4.2 Example code

With the C++ API, a simple program could look like:

```
#include "EventIO.hh"
using namespace eventio;

int main()
{
   int32_t data[2] = { 17, 10815 };
   EventIO iobuf;
   iobuf.OpenOutput("output.file");
   EventIO::Item item(iobuf,"put",99,0,123);
   if ( item.Status() != 0 )
      return 1;
   item.PutInt32(data,2);
   item.Done();
   iobuf.CloseOutput();
   return 0;
}
```

This simple program should write an item of type number 99, version number 0, and ID number 123 to a file. In this case, the item only contains two integers. For a nested item tree, it could look like:

```
#include "EventIO.hh"
using namespace eventio;

int main()
{
   int32_t data1[2] = { 17, 10815 };
   std::string data2("some text");
   EventIO iobuf;
   iobuf.OpenOutput("output.file");
   EventIO::Item item(iobuf,"put",101);
   if ( item.Status() != 0 )
      return 1;

   EventIO::Item item1(item,"put",102);
   item1.PutInt32(data1,2);
   item1.Done();

   EventIO::Item item2(item,"put",103);
   item2.PutString(data2);
   item2.Done();

   item.Done();
   iobuf.CloseOutput();
   return 0;
}
```

4.3 The EventIO and EventIO::Item classes

All of the internals of the `IO_BUFFER` are hidden in the C++ class `EventIO` which, in its constructor and destructor, takes care of the essential initialisation and cleanup. The EventIO class provides methods to open and close input files or functions, locating top-level blocks in the input stream and reading them.

The `EventIO::Item` sub-class is used to get all the items and data into the buffer or get them from the buffer. If the `EventIO::Item` constructor is called with an `EventIO` argument, a top-level item will be created (after finishing any incomplete operations of the buffer). If it is called with a `EventIO::Item` argument, the new item will be a sub-item of the given argument. The whole tree of such items can be created either for `"get"` or for `"put"` operations.

4.4 EventIO methods

The methods provided by the EventIO class can be subdivided into two groups:

1. Methods for connection input/output files or functions. These are mainly the `OpenInput` and `OpenOutput` methods (which take either a file name or a `FILE *` pointer as their argument), the `OpenFunction` method taking a `IO_USER_FUNCTION` argument, as well as the corresponding `CloseInput`, `CloseOutput`, and `CloseFunction` methods (all without arguments). All of them return 0 on success and -1 or other negative numbers on failure.

2. The `Find()`, `Read()`, `Skip()`, and `List()` methods correspond to the `find_io_block`, `read_io_block`, `skip_io_block`, and `list_io_block` C functions. After a `Find()` or `Read()`, you can also use the `ItemType()`, `ItemVersion()`, and `ItemIdent()` methods to inspect the properties of the current top-level item.

For low-level access to C methods, the underlying `IO_BUFFER` can be obtained through the `Buffer()` method.

4.5 Data access methods

For an `EventIO::Item` constructed with a `"get"` argument, data can be retrieved with the `GetXyz(...)` type functions. For an `EventIO::Item` constructed with a `"put"` argument, data can be written to the buffer with the `PutXyz(...)` type functions. In either case, an item should normally be finished by the `Done()` method but will be finished also by the end of its lifetime or by conflicting operations on its parent item, like creating a new `EventIO::Item` item at the same level as the current item or higher up in the hierarchy, or calling the `Done()` method at a higher level. The `Xyz` part here stands for the different data types:

- `Uint8` (Byte in C API)
- `Count`
- `SCount`
- `Int16` (Short in C API)
- `UInt16`
- `Int32` (Long in C API)
- `UInt32`
- `Int64`
- `Uint64`
- `Real`
- `Double`

- `String` (VarString in C API)

There are no methods directly corresponding to the older C data type String and LongString but these are easy to emulate where compatibility with older C code is required (see the `TestIO` program, for example).

All the `GetXyz(...)` and `PutXyz(...)` type functions (with exception of strings) come in a number of flavours:

- for single values,
- for arrays of values,
- for `vector` of values, and
- for `valarray` of values.

With `vector`, `valarray`, as well as with the special `string` methods, the corresponding header files must be included **before** including `EventIO.hh`. In addition, the `vector` and `valarray` methods come in two variants:

- `PutXyz(vec, nelem);`
- `PutXyz(vec);`

The first variant is analogous to the array flavour and only writes the contents of the array/vector/valarray. It assumes that the length of it is stored separately. The second variant, not available with arrays, will first put the number of elements to be stored into the buffer (as a `Count` type), and then all of its data elements. It is therefore the same if you write

```
PutCount(vec.size());
PutXyz(vec,vec.size());
```

or

```
PutXyz(vec);
```

The same holds true for the corresponding `GetXyz` methods. The variant without explicit length assumes that the element count precedes the data, the other only reads the given number of data elements. With `vector` and `valarray`, they will be resized to hold all of the data (in the variant without explicit length always, in the other variant only when the data would not fit). In the C API or with the array type flavour, this is not possible. There you always have to provide buffers of suitable length.

A special variant of array/vector/valarray methods is available for the SCount data type. If arrays of similar numbers have to be stored, it is often more efficient to store them as differences to the preceding one, only writing the absolute value for the first element. These methods are called `PutDiffSCount` and `GetDiffSCount`, with the same output for

```
PutDiffSCount(vec);
```

or

```
PutCount(vec.size());
if ( vec.size() > 0 )
   PutSCount(vec[0]);
for (size_t i=1; i<vec.size(); i++)
   PutSCount(vec[i]-vec[i-1]);
```

Appendix

A Object formats (examples)

A.1 Histograms as implemented by histogram.c/io_histogram.c

Object type:	100
Version:	2
ID:	ID no. of the first histogram or -1

Variable	Type	Count	Description
`nhisto`	short	1	Number of histograms following
Per Histogram:			
`type`	byte	1	Histogram type ('I', 'i', 'R', 'r', 'F', 'D')
`title`	string	1	Histogram title
—	byte	0 oder 1	fill byte if `type` and `title` together are of odd length.
`ident`	long	1	Histogram ID no.
`nbins`	short	1	No. of intervals (in x)
`nbins_2d`	short	1	No. of intervals in y or 0
`entries`	long	1	Total number of entries
`tentries`	long	1	Number of entries within limits
`underflow`	long	1	Entries below lower limit in x
`overflow`	long	1	Entries above upper limit in x
For Histograms with floating point boundaries (Types 'R', 'r', 'F' and 'D'):			
`lower_limit`	real	1	Lower limit (in x)
`upper_limit`	real	1	Upper limit (in x)
`sum`	real	1	Sum of all entered x values and
`tsum`	real	1	the same within the limits.
For Histograms with integer boundaries (Types 'I' and 'i'):			
`lower_limit`	long	1	Lower limit (in x)
`upper_limit`	long	1	Upper limit (in x)
`sum`	long	1	Sum of all entered x values and
`tsum`	long	1	the same within the limits.
For 2-D histograms only (`nbins_2d > 0`):			
`underflow_2d`	long	1	Entries below `lower_limit_2d`
`overflow_2d`	long	1	Entries above `upper_limit_2d`
For Histograms with floating point boundaries (Types 'R', 'r', 'F' and 'D'):			
`lower_limit_2d`	real	1	Lower limit in y
`upper_limit_2d`	real	1	Upper limit in y
`sum_2d`	real	1	Not used
`tsum_2d`	real	1	Not used
For Histograms with integer boundaries (Types 'I' and 'i'):			
`lower_limit_2d`	long	1	Lower limit in y
`upper_limit_2d`	long	1	Upper limit in y
`sum_2d`	long	1	Sum of all y values and
`tsum_2d`	long	1	the same within the limits.
Only for Histograms with integer contents (Types 'I', 'i', 'R' and 'r'):			
`counts`	long	(`nbins`) (`*nbins_2d`)	Contents if `tentries > 0` (1D) or 2-D Histogram (row by row)
Only for Histograms with weighted contents (Types 'F' and 'D'):			
`content_all`	real	1	Sum of all contents
`content_inside`	real	1	Sum of all within limits
`content_outside`	real	8	and in sectors outside
`data`	real	(`nbins`) (`*nbins_2d`)	Histogram contents if `tentries > 0` (1D) or 2-D Histogram (row by row)

B Internal C Structures

From the internal data structures, only those elements are being listed here that are intended to be used by application programs (omissions marked as '...'). Concerning terminology for beginners in 'C', it should be noted that a declaration like

```
struct _struct_SOMETHING { ... };
typedef struct _struct_SOMETHING SOMETHING;
```

or short

```
typedef struct { ... } SOMETHING;
```

requires a definition in the application program as

```
SOMETHING name;
```

In the following, the shorter form will be used, although in the code the first and longer form is used.

B.1 Structures for I/O and data object management

See also section 3.1.

```
typedef struct
{
   ...
   int input_fileno;
   int output_fileno;
   FILE *input_file;
   FILE *output_file;
   int (*user_function)(unsigned char *, long, int);
   ...
   long max_length;
   ...
}  IO_BUFFER;


typedef struct
{
   ...
   unsigned long type;
   unsigned version;
   long ident;
   ...
}  IO_ITEM_HEADER;
```