Serial

Input

Output

0 What Is SIO?

Serial Input/Output (SIO) is designed to be a long term storage format of a sophistication somewhere between simple ASCII files and the techniques provided by *inter alia* Objectivity and Root. The former tend to be low density, information lossy (floating point numbers lose precision) and inflexible. The latter require abstract descriptions of the data with all that that implies in terms of extra complexity.

0.1 Features Provided by SIO

Architecture independent binary format.

A high integrity, self-checking data layout.

Multiple simultaneously open input and output streams.

Heterogeneous record types on each stream.

Pointer relocation at the level of a record.

On the fly data compression/decompression.

0.2 Features NOT Provided by SIO

Abstract data descriptions (i.e. self describing data).

Pointer chasing.

1 Technical Overview

1.1 Building Blocks

The basic building blocks of SIO are streams, records and blocks.

Streams provide the connections between the program and files. The user can define an arbitrary list of streams as required. A given stream must be opened for either reading or writing. SIO does not support read/write streams. If a stream is closed during the execution of a program, it can be reopened in either read or write mode to the same or a different file.

Records represent a coherent grouping of data. Records consist of a collection of blocks (see next paragraph). The user can define a variety of records (headers, events, error logs, etc.) and request that any of them be written to any stream. When SIO reads a file, it first decodes the record name and if that record has been defined and unpacking has been requested for it, SIO proceeds to unpack the blocks.

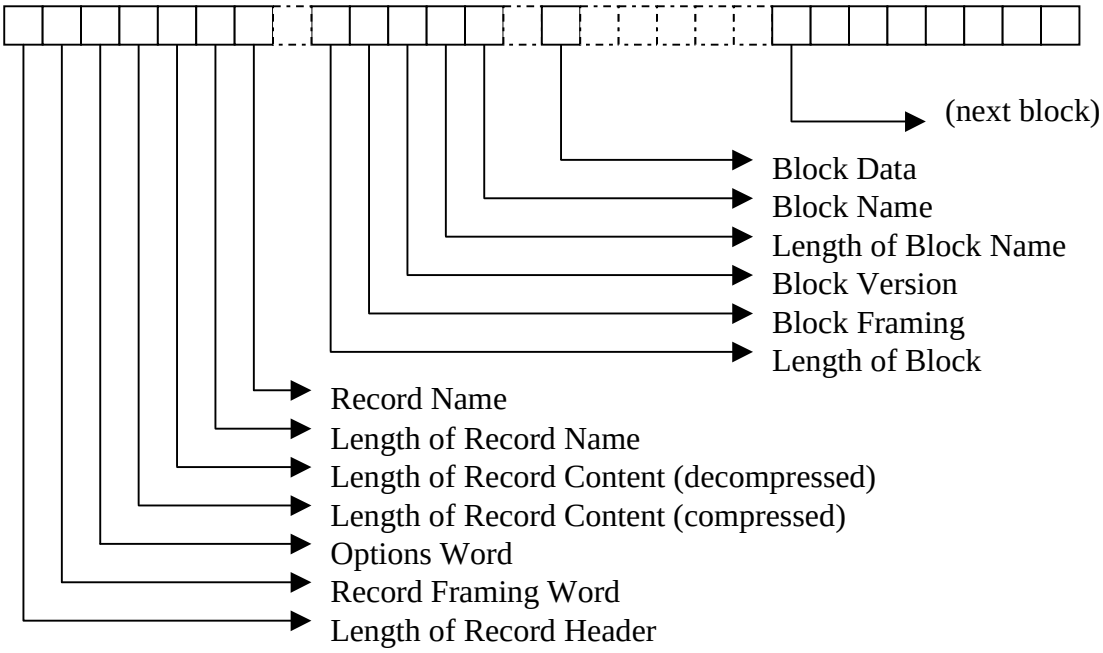
Blocks are user provided objects which do the real work of reading/writing the data. The user is responsible for writing the code for these blocks and for identifying these blocks to SIO at run time. To write a collection of blocks, the user must first connect them to a record. The record can then be written to a stream as described above. Note that the same block can be connected to many different records. When SIO reads a record, it scans through the blocks written and calls the corresponding block object (if it has been defined) to decode it. Undefined blocks are skipped.

Each of these categories (streams, records and blocks) have some characteristics in common. Every stream, record and block has a name with the condition that each stream, record or block name must be unique in its category (i.e. all streams must have different names, but a stream can have the same name as a record).

Each category is an arbitrary length list which is handled by a ‘manager’ and there is one manager for each category.

1.2 Navigation Format

The above definitions of streams, records and blocks allow SIO to write ‘navigable’ datasets. Ignoring details of data representation and the contents of the blocks, an SIO dataset is composed of a series of records which look like (each box represents four bytes):



- Length of Record Header. Four bytes. Length of the record header in bytes (including the four bytes needed for this number). Always a multiple of four.
- Record Framing. Four bytes. A weird hex value (actually `0xabadcafe`) which tries to identify what follows as a record. Not foolproof, but pretty good.
- Options Word. Four bytes. Reserved for SIO. Currently only the least significant bit is used. When set, the record content that follows has been compressed.
- Length of Record Content (compressed). Four bytes. Total length in bytes for all the blocks in the record. Always a multiple of four. If the record is not compressed, it contains the same value as…
- Length of Record Content (decompressed). Four bytes. Total length in bytes for all the blocks in the record *when decompressed*. Always a multiple of four. When the record is not compressed, it is a count of the bytes that follow in the

record content. When the record is compressed, this number is used to allocate a buffer into which the record is decompressed.

- Length of Record Name. Four bytes. Length of the record name specified by the user. This is a *character* count and can take any value.
- Record Name. Arbitrary length, but padded with zeros to a four byte boundary. Standard ASCII encoding. Name of the record (surprise!).
- Length of Block. Four bytes. Length of the block in bytes (including the four bytes needed for this number).
- Block Framing. Four bytes. Another weird hex value (actually `0xdeadbeef`) which tries to identify what follows as a block. Same caveats as 'Record Framing'.
- Block Version. Four bytes. My attempt to enforce some versioning information. All blocks must provide a version number to allow block evolution while preserving readability of old datasets.
- Length of Block Name. Four bytes. Length of the block name specified by the user. This is again a *character* count and can take any value.
- Block Name. Arbitrary length, but padded with zeros to a four byte boundary. Standard ASCII encoding. Name of the block (surprise!).
- Block Data. Arbitrary length. The data written/read by the user's block object.

With this format, SIO can read down the dataset skipping blocks or records if they're not requested. It can also detect and recover from data overruns or underruns by the user provided block readers (though continuing to work with a record after such an error is highly questionable).

1.3 Data Representation

The previous section carefully ignored the representation of (for instance) the four bytes giving the 'Length of Record Header'. That's because I didn't want to get sidetracked into a discussion of endian-ness while discussing navigation. Endian-ness will not be ignored however, so I invite you to consider the following program:

```
#include <stdio.h>

struct _buffer {
    union {
        unsigned char  u_byte[4];
        unsigned int   u_int;
    } data;
} buffer;

int main()
{
    int i;

    for( i = 0; i < 4; i++ )
    {
        buffer.data.u_byte[i] = i;
    }


    printf( "0x%08x", buffer.data.u_int );
    return 0;
}
```

Try this on a big-endian architecture (PowerPC (AIX), sparc (SunOS)) and you will get `0x00010203`. Try it on a little-endian architecture (alpha (DEC/OSF1), x86 (Linux or Windows/NT)) and you'll get `0x03020100`. Not good when trying to construct datasets readable by all architectures!

There were, and possibly still are, middle-endian architectures! A wholly mind-boggling concept matched only by the oxymoronic name.

Swiftian wars are of course fought between those that favour one endian-ness over the other. Whatever the merits of their cases, SIO has to deal with both varieties. The SIO solution is to adopt the `xdr` format. If you look `xdr` up on the web you will find many learned documents both defining the format and discussing its merits. For our purposes, `xdr` can be summarized with two rules:

- `xdr` is always big-endian.
- Any single `xdr` write will pad with zeros up to the next four byte boundary (and equivalently, `xdr` reads will skip forward to the next four byte boundary).

Voila, everything you need to know about `xdr`!

Users should be aware of that second rule. Ignorance can lead to large inefficiencies. The following examples will use SIO techniques which haven't been described yet, but the intent should be clear:

```
int         i;
int         list[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

.
.
for( i = 0; i < 10; i++ )
{
    SIO_DATA( stream, &list[i], 1 );
}
.
.
```

Example 1

```
int         list[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

.
.
SIO_DATA( stream, &list[0], 10 );
.
.
```

Example 2

Example 1 uses 10 writes to output 10 bytes, but each byte gets padded to the next four byte boundary for a total of 40 bytes. Example 2 uses 1 write to output 10 bytes which then gets padded out to 12 bytes. A factor of three saving!

2 Coding SIO Blocks

User blocks are coded as descendants of the abstract class `SIO_block`. In the implementation of a new block, the user must override the pure virtual methods `version` and `xfer and` call the constructor for the base `SIO_block.` After adding SIO capability to an existing object `foo`, `foo.h` might look something like this:

```
.
.
#include “SIO_block.h”
class SIO_stream;
.
.
class foo : public SIO_block
{

public:
    //
    // Add an extra argument to the foo constructor to take the block name.
    //
    foo(..., const char* );
    .
    .
    //
    // SIO required functions.
    //
    unsigned int version();
    unsigned int xfer( SIO_stream*, SIO_operation, unsigned int );

private:
    .
    .
};
```

And `foo.cxx` something like:

```
.
.
#include “foo.h”
.
#include “SIO_definitions.h”
#include “SIO_functions.h”
.
.
// ---------------------------------------------------------------------------
// Constructor.
// ---------------------------------------------------------------------------
foo::foo(..., const char* i_name ) : SIO_block( i_name )
{
.
// <standard constructor stuff (nothing to do with SIO)>
.
}

// ---------------------------------------------------------------------------
// Perform SIO transfer.
// ---------------------------------------------------------------------------
unsigned int foo::xfer
(
    SIO_stream*     stream,
    SIO_operation   op,
    unsigned int    version
)
{
.
// <SIO commands to transfer data (not yet described)>
.
return( SIO_BLOCK_SUCCESS );
}

// ---------------------------------------------------------------------------
// Return SIO version number.
// ---------------------------------------------------------------------------

#define SIO_FOO_MAJOR 1
#define SIO_FOO_MINOR 0

// <note the use of an SIO macro to encode a major and a minor version>

unsigned int foo::version()
{ return( SIO_VERSION_ENCODE( SIO_FOO_MAJOR, SIO_FOO_MINOR ) ); }
```

Of course this isn’t the only way to introduce SIO. If you don’t want to interfere with existing classes, you could write a class `SIO_foo` which accesses `foo` data through ‘get’ methods, or alternatively declare `SIO_foo` a friend in `foo`.

2.1 The 'version' Method

The `version` method should be reasonably self explanatory. It returns an `unsigned int` which describes the version level of the block. For convenience, I provide the `SIO_VERSION_ENCODE` macro to encode a major and a minor value (each in the range 0 - (2^{16}-1). Other macros (`SIO_VERSION_MAJOR` and `SIO_VERSION_MINOR`) are provided to reverse the encoding process. It is not mandatory for a block to follow a major/minor versioning scheme. It *is* mandatory for a block to have a `version` method returning an `unsigned int`.

2.2 The 'xfer' Method

This is where the fun really begins! First of all, why `xfer`? Why not a `read` method and a `write` method? Because if you're careful, *you can use the same method!*

The formal parameters to the `xfer` method are:

- `SIO_stream*` `stream`

  An opaque pointer which you just pass on to the data transfer routines.

- `SIO_operation` `op`

  An enumerated value. Either `SIO_OP_READ` (when the `xfer` method is being asked to read from the stream) or `SIO_OP_WRITE` (when the `xfer` method is being asked to write to the stream).

- `unsigned int` `version`

  The version described in the previous section. When writing, it is the value returned by a call to the `version` method. When reading it is the block version number read from the data stream.

Inside `xfer`, the workhorse routine to transfer data to and from a stream is `SIO_data`. `SIO_data` expects to be called with three parameters:

- `SIO_stream*` `stream`

  Just pass the value passed in to xfer as the first parameter.

- `<pointer to ?>` `pntr`

  A pointer to a location in memory where data transfer should start. I'm afraid I lied when I said `SIO_data` (singular) was the data transfer workhorse. In fact, there are many copies of `SIO_data`, each one taking a different primitive data type. Thus there is a version of `SIO_data` which accepts a pointer to short

and another that accepts a pointer to double. The complete set of recognized primitives are:

```
         char
unsigned char
         short
unsigned short
         int
unsigned int
         long long
unsigned long long
float
double
```

Notable by its absence is `long`. Unfortunately `long` means different things on different architectures (on DEC/OSF1/Alpha it's a 64 bit integer). What does it mean to write a `long` (64 bits) on DEC/OSF1/Alpha and then read it back as a `long` (32 bits) on SunOS/sparc? I couldn't figure it out so I simply didn't provide an interface.

The version of M$ Visual Studio I'm using (C++ 5.0) doesn't understand the ANSI specified declaration `long long` for a 64 bit integer (it insists on its own invention called `__int64`). If you want to use a 64 bit integer, you should for now declare it as `SIO_64BITINT`.

Stop Press. The same is true in M$ Visual Studio 6.0

- `int count`

  The number of primitives to be transferred.

2.2.1 A Simple Write Routine

With these definitions let's write a notional piece of code to write out a run number, an event number and floating point three vector called `position`.

```
static int       run      = 1;
static int       event    = 42;
static double    position = { 1.0, 2.0, 3.0 };
unsigned int     status;
    .
    .
    status = SIO_data( stream, &run,         1 );
    status = SIO_data( stream, &event,       1 );
    status = SIO_data( stream, &position[0], 3 );
    .
    .
```

So far so good. But notice that `SIO_data` returns an `unsigned int`. This is an error code (yes Virginia, errors do occur … what if you suddenly lost your AFS token for instance). The error code follows the DEC/VMS principles of encoding a variety of information in bit fields, but the only piece of encoding of interest to the general user is that if the least significant bit is set, the operation was successful. Revisiting our notional code and doing the proper error handling:

```
static int       run      = 1;
static int       event    = 42;
static double    position = { 1.0, 2.0, 3.0 };
unsigned int     status;
    .
    .
    status = SIO_data( stream, &run,         1 );
    if( !(status & 1) )
        return( status );

    status = SIO_data( stream, &event,       1 );
    if( !(status & 1) )
        return( status );

    status = SIO_data( stream, &position[0], 3 );
    if( !(status & 1) )
        return( status );

    .
    .
```

Well the error handling is correct, but the code is getting ugly! To help out, I provide yet another macro (called `SIO_DATA`) which does all this work for you. The code, complete with error handling, can be written thus:

```
static int       run      = 1;
static int       event    = 42;
static double    position = { 1.0, 2.0, 3.0 };
unsigned int     status;
    .
    .
    SIO_DATA( stream, &run,         1 );
    SIO_DATA( stream, &event,       1 );
    SIO_DATA( stream, &position[0], 3 );
    .
    .
```

The code is certainly neater, but there is a hidden gotcha. If you want to use the `SIO_DATA` macro, you *must* declare an `unsigned int` called `status`.

2.2.2 The Corresponding Read Routine

Here it is:

```
static int       run      = 1;
static int       event    = 42;
static double    position = { 1.0, 2.0, 3.0 };
unsigned int     status;
    .
    .
    SIO_DATA( stream, &run,         1 );
    SIO_DATA( stream, &event,       1 );
    SIO_DATA( stream, &position[0], 3 );
    .
    .
```

Look familiar? No I didn't make a mistake with cut and paste! In this example the code to read back the data is *identical* to the code needed to write it. It's true that all those initialization values will get overwritten during the read, and I should probably have omitted them, but I wanted to reinforce the point that reading and writing can be accomplished by a very similar, if not the identical piece of code.

2.3 The Real World

Real world SIO blocks will obviously be much more complicated than the simple example just outlined. There will be times when it is impossible to write exactly identical code for the read and the write. Some of these can be overcome by minor code branching based on the value of `op` (the second parameter to `xfer`). In extreme examples, it may be necessary to write your own private and separated read and write routines to which `xfer` branches.

Another real world effect is data format evolution. Much can be achieved using the version number. Real `xfer` routines may well end up looking like:

```
unsigned int MyClass::xfer
(
    SIO_stream*     stream,
    SIO_operation   op,
    unsigned int    version
)
{
.
.
.
    major = SIO_DECODE( version );
    minor = SIO_DECODE( version );

    switch( major )
    {
        case 1:
            SIO_DATA( stream, &run,         1 ); // Original major=1 format
            SIO_DATA( stream, &event,       1 ); // Original major=1 format
            SIO_DATA( stream, &position[0], 3 ); // Original major=1 format
            new_data = <guard_value>;            // Fake data when major=2
            break;

        case 2:
            SIO_DATA( stream, &run,         1 ); // Common to major=1, major=2
            SIO_DATA( stream, &event,       1 ); // Common to major=1, major=2
            SIO_DATA( stream, &position[0], 3 ); // Common to major=1, major=2
            SIO_DATA( stream, &new_data,    1 ); // New data in major=2 format
            break;
    }
.
.
.
return( SIO_BLOCK_SUCCESS );
}
```

When this routine is used to write data, it is still called with a version number. The version number is whatever the `version` member of the class currently responds with, which is presumably the latest evolution (`major=2` in this case). When this routine is used to read current data (`major=2`) then all is well. When it is used to read old data (`major=1`) the data can still be read, but the value of `new_data` has to be faked (hopefully with a unique and identifiable guard value).

3 Pointers

One of the more frustrating aspects of constructing datasets for C or C++ is the problem of what to do about pointers. Memory references which made sense in the context of a program are totally meaningless when written to a file.

More sophisticated tools like Root, Objectivity and Java provide methods for dealing with this problem. Those methods can be quite complex, involving pointer chasing (the idea that if you write a pointer, you also want to write the object at the far end of the pointer) and duplicate elimination (more than one pointer might point to the same object, which should not be written out twice for efficiency reasons). This style also requires genuinely self describing objects (because the program must be able to work out how to write the object at the far end of a pointer).

SIO does not try to compete at this level, which is not to say that SIO doesn't provide some help! Lacking any form of data description, SIO *cannot* pointer chase. This puts the responsibility back in the programmer's court. If object A contains a pointer to object B, then it is up to the programmer(s) to ensure that objects A and B are each written out. With that proviso, SIO can do the rest.

3.1 Pointer To and Pointed At

As SIO constructs a record to be written out, it takes special note of data which the user declares to be either a 'pointer *to* object' or a 'pointed *at* object' (the next section will give details of how the user can do that). These are stored in lookaside tables and just before physically writing the data out, SIO goes back over the record and enters 'match values' such that every 'pointed at object' gets a unique identifier and all 'pointer(s) to (that) object' get the same identifier.

When this record is read back, SIO once again constructs lookaside tables which take note of where the 'pointers to objects' and 'pointed at objects' end up in

memory. When record reading is complete, it does a relocation pass to fix up all the pointers.

3.2 Pointer Example

So much for theory. How is this accomplished in a real program? The following is the code I used to test this feature. It is that ubiquitous data structure, the singly linked list. The first listing is the header file and the second listing is the implementation file.

```
// ------------------------------------------------------------------------
// => Test SIO on a linked list.
// ------------------------------------------------------------------------
//
// General Description:
//
// SIO_list_test tests the SIO formatting of a linked list.
//
// ------------------------------------------------------------------------

#ifndef SIO_LIST_TEST_H
#define SIO_LIST_TEST_H 1

#include "SIO_block.h"

class SIO_stream;

typedef struct s_LinkList {
    struct s_LinkList*   next;
    float                value;
} LinkList;

class SIO_list_test : public SIO_block
{
public:
    SIO_list_test( const char* );

    unsigned int           xfer( SIO_stream*, SIO_operation, unsigned int );
    void                   validate();
    unsigned int           version();

private:
    LinkList*              link_list;
    LinkList*              link_read;

};
#endif
```

```
// ---------------------------------------------------------------------------
// => Test SIO on a linked list.
// ---------------------------------------------------------------------------
//
// General Description:
//
// SIO_list_test tests the SIO formatting of a linked list.
//
// ---------------------------------------------------------------------------

#ifdef _MSC_VER
#   pragma warning(disable:4786)        // >255 characters in debug information
#endif

#include <stdio.h>

#include "SIO_list_test.h"
#include "SIO_definitions.h"
#include "SIO_functions.h"

// ---------------------------------------------------------------------------
// Constructor.
// ---------------------------------------------------------------------------
SIO_list_test::SIO_list_test( const char* i_name ) : SIO_block( i_name )
{

//
// Local variables.
//
LinkList
   *curr,
  **prev;

int
    i;

//
// Initialize the read in pointer.
//
link_read = NULL;

//
// Build a dumb linked list.
//
prev = &link_list;
for( i = 0; i < 4; i++ )
{
    curr  = static_cast<LinkList *>(malloc( sizeof( LinkList ) ));
    curr->next = NULL;
    curr->value = 0.1 * (i + 1);
    *prev = curr;
    prev  = &(curr->next);
}
```

```
//
// That's all folks!
//
}

// ---------------------------------------------------------------------------
// Transfer.
// ---------------------------------------------------------------------------
unsigned int SIO_list_test::xfer
(
    SIO_stream*     stream,
    SIO_operation   op,
    unsigned int    version
)
{
LinkList
   *curr;

unsigned int
    status;

//
// Do the transfer.
//
if( op == SIO_OP_READ )
{
    SIO_PTAG( stream, &link_read );
    SIO_PNTR( stream, &link_read );
    curr = link_read;
}
else
{
    SIO_PTAG( stream, &link_list );
    SIO_PNTR( stream, &link_list );
    curr = link_list;
}

while( curr != NULL )
{
    if( op == SIO_OP_READ )
        curr = static_cast<LinkList *>(malloc( sizeof( LinkList ) ));

    SIO_PTAG( stream, &curr->next     );
    SIO_PNTR( stream, &curr->next     );
    SIO_DATA( stream, &curr->value, 1 );

    if( op == SIO_OP_READ )
    {
        if( curr->next == NULL )
            curr = NULL;
    }
    else
        curr = curr->next;
```

```
}

//
// That's all folks!
//
return( SIO_BLOCK_SUCCESS );
}

// ---------------------------------------------------------------------------
// Validate.
// ---------------------------------------------------------------------------
void SIO_list_test::validate()
{
//
// Local variables.
//
LinkList
   *curr;

//
// Validate the data content.
//
printf( "\n---------- In memory ----------\n" );
for( curr = link_list; curr != NULL; curr = curr->next )
{
    printf( "0x%016lx:  0x%016lx  0x%016lx  %8.6f\n",
        curr, curr->next, curr->head, curr->value );
}

printf( "\n---------- Read back ----------\n" );
for( curr = link_read; curr != NULL; curr = curr->next )
{
    printf( "0x%016lx:  0x%016lx  0x%016lx  %8.6f\n",
        curr, curr->next, curr->head, curr->value );

}

//
// That's all folks!
//
return;
}

// ---------------------------------------------------------------------------
// Return version number (encoded).
// ---------------------------------------------------------------------------
#define SIO_LIST_TEST_MAJOR 1
#define SIO_LIST_TEST_MINOR 0

unsigned int SIO_list_test::version()
{ return( SIO_VERSION_ENCODE( SIO_LIST_TEST_MAJOR, SIO_LIST_TEST_MINOR ) ); }
```

That's quite a mouthful, so I'll try to explain what's going on:

- The constructor always constructs a simple four member linked list. The 'root' pointer is the variable `link_list`. This is the data that gets written out.
- The constructor also initializes the `link_read` variable to `NULL`. In order to do the test, I wanted to have both the data written out and the data read back in memory simultaneously so that I could compare them. This is going to complicate…
- The `xfer` routine does both the reading and the writing. Because the read and written versions of the data occupy different places in memory, I had to use the `op` variable to branch portions of the reading and writing (though in truth, even if the data was read back into the same location, it is the nature of this example that I would still need to use `op`).
- The `verify` routine simply prints the structures out to see if SIO worked.
- When I wanted to mark something as being 'pointed at', I used the `SIO_PTAG` macro which takes a two arguments:
  - The usual opaque pointer (`stream`)
  - The address in memory I wish SIO to remember (in this case the variable `next` is the first element of the structure `LinkList`, so `&curr->next` is just the address of that structure in the linked list).
- When I wanted to mark something as being a 'pointer to', I used the `SIO_PNTR` macro, which also takes two arguments:
  - The usual opaque pointer (`stream`)
  - The address in memory which *contains* the pointer I wish SIO to remember (in this case `curr->next` is the pointer, so the address that *contains* this pointer is `&curr->next`).

At which point I must apologize for coming up with an example where the arguments to `SIO_PTAG` and `SIO_PNTR` are identical. This only happens when the something being pointed to is itself a pointer to something else!

3.2.1 Pointer Length

If you look at the dump routine, you will find the formatting instruction `0x%016lx` being used to dump pointers. Why? Because the length of a pointer is not guaranteed by either C or C++. In fact pointers are four bytes long on IBM/AIX, Linux, Sun/SunOS and Windows/NT. They are *eight* bytes on DEC/OSF1!

3.2.2 NULL pointer handling

There is one other feature of this example which is very important. If you look at the code executed during a read, you will notice that the value of `curr->next` (a pointer) is used immediately after it is read. How can this be legal when SIO has not yet done its pointer relocation?

The answer is that `NULL` pointers are special. If `SIO_PNTR` is asked to write out a `NULL` pointer, SIO guarantees that a `NULL` pointer will be returned immediately when the data is read back. Conversely, `SIO_PNTR` always returns a non `NULL` value for a pointer which was non `NULL` when it was written out (though the non `NULL` value `SIO_PNTR` returns is otherwise useless until after SIO has done its relocation pass).

3.3 Good Points About SIO Pointer Handling

- First of all that it exists! At least you get some help with pointers!
- SIO defines a consistent representation of pointers in the dataset, thus relieving the user from concerns about the exact length of pointers.
- The relocation pass doesn't occur until all the blocks of a record have been read or written so pointers can point to any location within the record. They are *not* constrained to point to objects which have already been written out.
- It's generally well behaved. If you `SIO_PTAG` something that nothing else points to, no harm is done. More importantly, if you `SIO_PNTR` to something that doesn't make it into the written record, that pointer will be set to `NULL` by the relocation. (This case is distinct from the point discussed earlier which only dealt with the case of what happens to a pointer that was `NULL` when it was written out).

3.3 Ambiguous Points About SIO Pointer Handling

- Pointer relocation is handled across a whole record. This is good in that pointers are not constrained to point within their own block. This can be bad if pointers in one author's block point to objects in another author's block. These two authors will need to get together to ensure that both their blocks appear in the written record.

3.3 Bad Points About SIO Pointer Handling

- SIO's method of relocation 'steals' memory locations during the read process, so the user code for an SIO block must always read the data into 'final' locations. Do *not* read data into temporary variables allocated from the stack!

SIO will cheerfully overwrite those locations, mess up the stack and leave you with a debugging problem you do not want to deal with!

4 Class Descriptions

4.1 Common Features

Before launching into a formal description of the SIO classes, I would like to describe some common features which run through all the classes. Setting these out first should make the SIO interface routines more comprehensible.

4.1.1 SIO Names

Names are used throughout SIO to identify streams, records and objects. A name must obey certain rules to be legal. Basically anything which would be legal as a C/C++ variable is a legal name. Formally that amounts to:

- Names must constructed from the characters in the upper- and lower-case alphabets, digits and the underscore.
- Names cannot begin with a digit.
- Names are case sensitive.

4.1.2 SIO Condition Codes

SIO frequently reports errors using a condition code (an `unsigned int`). These codes follow the old VMS standard of encoding data into bit fields. The only part of real interest to a user is that if the least significant bit of an error code is set, the operation was successful. All condition codes are defined in the file `SIO_definitions.h`.

4.1.2 SIO Enumerations

Many parameters to and returns from SIO routines are defined as members of enumerated sets. For example, when a stream is opened to a file, the second argument is defined to be a member of the enumerated set `SIO_stream_mode`. Possible values of `SIO_stream_mode` are `SIO_MODE_READ`, `SIO_MODE_WRITE_NEW`, `SIO_MODE_WRITE_APPEND`. All enumerated sets are defined in `SIO_definitions.h`.

4.1.3 SIO Error Reporting

SIO has a flexible method of controlling the reporting of errors to `<stdout>`. It defines three levels of reporting using the enumerated set `SIO_verbosity`. The levels are:

- `SIO_SILENT` Never report anything.
- `SIO_ERRORS` Report errors.
- `SIO_ALL` Report errors and progress.

The reporting level is not a single global variable. Verbosity can be defined down to the level of each individual stream, record or block. The technique is the same for all three, so I'll only describe it for streams.

When a stream is created, it receives its reporting level from the default level established in the stream manager. A stream's reporting level can then be tailored using that stream's `getVerbosity` and `setVerbosity` methods. The manager's default level can also be tailored using the manager's `getVerbosity` and `setVerbosity` methods. This will alter the reporting level of all streams created *after* the change in the manager's default. If the user chooses not to tailor the manager's reporting level, the program default is `SIO_ERRORS`.

The format of reported errors is also designed to be consistent. If the block manager wishes to report an error it will be in the form:

```
SIO: [Block Manager] <Text of error>
```

If a block wishes to report an error, in will be in the form:

```
SIO: [<stream>/<record>/<block>] <Text of error>
```

4.2 SIO Managers

The SIO managers are very similar. In essence they simply maintain lists of named objects of their own 'species'. They are never instantiated and all their methods and data are declared static. You can think of them as singleton objects.

4.2.1 SIO_streamManager Methods

`add( const char* name )`

`name`	Input	Name of stream.
	Returns	`SIO_stream*`

Create a new stream. At time of creation, it is not necessary to specify how the stream will be used. In case of error (illegal name for instance), it returns a `NULL` pointer.

`add( const char* name, unsigned int size )`

`name`	Input	Name of stream.
`size`	Input	Suggested stream buffer size (in bytes)
	Returns	`SIO_stream*`

Variant on the previous `add`. You can help SIO by suggesting the right sort of buffer size to allocate for transferring these records. If you can't it doesn't matter, SIO will look after its own buffering.

`get( const char* name )`

`name`	Input	Name of stream.
	Returns	`SIO_stream*`

Return a pointer to the named stream. If the stream does not exist, the `NULL` pointer is returned.

`getVerbosity()`

	Returns	`SIO_verbosity`

Return the manager's default verbosity setting.

`remove( const char* name )`

`name`	Input	Name of stream.
	Returns	`unsigned int` (condition code)

Remove a stream. If the stream is open, it will be closed first.

`setVerbosity( SIO_verbosity verbosity )`

`verbosity`	Input	New default verbosity
	Returns	`SIO_verbosity`

Set the manager's default verbosity. The value returned is the verbosity before the change.

4.2.2 SIO_recordManager Methods

`add( const char* name )`

`name`	Input	Name of record.
	Returns	`SIO_record*`

Create a new record. If the record cannot be created, the `NULL` pointer is returned.

`get( const char* name )`

`name`	Input	Name of record.
	Returns	`SIO_record*`

Return a pointer to the named record. If the record does not exist, the `NULL` pointer is returned.

`getVerbosity()`

	Returns	`SIO_verbosity`

Return the manager's default verbosity.

`remove( const char* name )`

`name`	Input	Name of record.
	Returns	`unsigned int (a condition code)`

Remove a record.

`setVerbosity( SIO_verbosity verbosity )`

`verbosity`	Input	New default verbosity
	Returns	`SIO_verbosity`

Set the manager's default verbosity. The value returned is the verbosity before the change.

4.2.3 SIO_blockManager Methods

`add( SIO_block* block )`

`block`	Input	Pointer to block object.
	Returns	`SIO_block*`

Add a pre-existing block to the list. This is not the same as `add` for streams and records. Streams and records are strictly SIO objects and SIO knows how to create and delete them. Blocks are instantiated outside SIO by the user and the pointer to the block is passed into this routine. If there is any problem with the addition, the `NULL` pointer is returned, otherwise it simply reflects its input parameter.

`get( const char* name )`

`name`	Input	Name of block.
	Returns	`SIO_block*`

Return a pointer to the named block. If the record does not exist, the `NULL` pointer is returned.

`getVerbosity()`

	Returns	`SIO_verbosity`

Return the manager's default verbosity.

`remove( const char* name )`

`name`	Input	Name of block.
	Returns	`unsigned int (a condition code)`

Remove a block.

`setVerbosity( SIO_verbosity verbosity )`

`verbosity`	Input	New default verbosity
	Returns	`SIO_verbosity`

Set the manager's default verbosity. The value returned is the verbosity before the change.

4.3 SIO_stream Methods

`close()`

	Returns	`unsigned int` (condition code)

Close the stream.

`dump( unsigned int offset, unsigned int length )`

`offset`	Input	Dump offset in the stream buffer
`length`	Input	Length of buffer to dump
	Returns	`void`

Produces a hex dump on `<stdout>` of the stream's buffer starting at byte offset `offset` and running for `length` bytes. This is essentially a debugging tool and is not for the squeamish.

`getName()`

	Returns	`std::string*`

Return this stream's name.

`getFilename()`

Returns `std::string*`

Return the name of the file associated with this stream. Blank if no file is associated with this stream.

`getMode()`

Returns `SIO_stream_mode`

Returns the stream mode. Mode will be a member of the enumerated set `SIO_stream_mode`. Modes include read (stream is set up to read from a file), write new (stream is set up to write to a file and will overwrite a pre-existing file of the same name), write append (stream is set up to write to a file and will append to a pre-exiting file of the same name) and undefined (the stream exists but has not yet been associated with a file).

`getState()`

Returns `SIO_stream_state`

Returns the stream state. State will be a member of the enumerated set `SIO_stream_state`. States include open (stream is attached to an open file), closed (stream is not attached to an open file) and error (something has gone wrong with this stream).

`getVerbosity()`

Returns `SIO_verbosity`

Return the verbosity level for this stream.

`open( const char* file, SIO_stream_mode mode )`

`file`	Input	Name of file to open.
`mode`	Input	Mode in which to open file.
	Returns	`unsigned int` (condition code)

Open a file on this stream. `mode` determines the mode (read, write new or write append).

`read( SIO_record** record )`

`record`	Output	Pointer to record type opened.
	Returns	`unsigned int` (condition code)

Read a record on the stream. The argument is provided because a stream can have many types of records on it. This allows the user to inspect the record type just read (by calling e.g. `record->getName`). Errors and the end-of-file condition are reported through the condition code.

`setVerbosity( SIO_verbosity verbosity )`

`verbosity`	Input	New verbosity
	Returns	`SIO_verbosity`

Set the verbosity for this stream. Returns the prior verbosity.

`write( const char* record )`

`record`	Input	Name of record to be written.
	Returns	`unsigned int` (condition code)

Write a record to the stream.

4.4 SIO_record Methods

`connect( const char* block )`

`block`	Input	Name of block to attach to record
	Returns	`unsigned int` (condition code)

Connect a block to a record. This is only needed when writing records. When a record is written, it will scan through and write each block connected to it. A block can be simultaneously connected to several records.

`connect( SIO_block* block )`

`block`	Input	Pointer to block to attach to record
	Returns	`unsigned int` (condition code)

Simply a variant interface. See previous entry.

`disconnect( const char* block )`

`block`	Input	Name of block to detach from record
	Returns	`unsigned int` (condition code)

Disconnect a block from a record. Blocks can be connected and disconnected at any time during the program (but beware, do you really want a record to have different content on the same data stream?)

`disconnect( SIO_block* block )`

`block`	Input	Pointer to block to detach from record
	Returns	`unsigned int` (condition code)

Simply a variant interface. See previous entry.

`getCompress()`

Returns `bool`

Get the current compression state for this record.

`getConnect( const char* block )`

`block`	Input	Name of block to query
	Returns	`SIO_block*`

Ask if the named block is connected to this record. Returns the block pointer if it is, a `NULL` pointer if it isn't.

`getName()`

Returns `std::string*`

Get the name of this record.

`getUnpack()`

Returns `bool`

Ask if record unpacking has been requested for this record. If record unpacking has not been requested then records of this name will be ignored when they are read.

`getVerbosity()`

Returns `SIO_verbosity`

Return the verbosity level for this record.

`setCompress( bool )`

Returns `bool`

Set the compression state for this record. Returns prior compression state.

`setUnpack( bool )`

Returns `bool`

Set the unpacking state for this record. Returns prior unpacking state.

`setVerbosity( SIO_verbosity verbosity )`

`verbosity`	Input	New verbosity
	Returns	`SIO_verbosity`

Set the verbosity for this record. Returns the prior verbosity.

4.5 SIO_block Methods

`getName()`

Returns `std::string*`

Get the name of the block.

`xfer( SIO_stream* st, SIO_operation op, unsigned int version )`

`st`	Input	Opaque stream pointer.
`op`	Input	Transfer operation (read or write).
`version`	Input	Block version number.
	Returns	`unsigned int` (condition code)

A pure virtual function. Discussed in the section 'Coding SIO Blocks'.

`version()`

Returns `unsigned int` (version number)

A pure virtual function. Discussed in the section 'Coding SIO Blocks'.

4.6 A Fake Example

Putting all this information together, what might a real program look like? The following example is trivial and is only pseudo-code, but it should give you some idea. The 'program' is reading an SIO dataset containing records named `raw_data` which in turn contain the blocks `Foo` and `Bar`. The program wants to munge on these blocks to produce a new block called `Baz`, at which point it wants to write out all three blocks in a single record called `baz_data` to a new SIO output file. I've embedded the rest of the documentation in the program comments:

```
//
// The required SIO includes.
//
#include "SIO_streamManager.h"
#include "SIO_recordManager.h"
#include "SIO_blockManager.h"
#include "SIO_stream.h"
#include "SIO_record.h"
#include "SIO_definitions.h"

//
// The includes defining Foo, Bar and Baz
//
#include "Foo.h"
#include "Bar.h"
#include "Baz.h"
```

```
//
// I'll need an input and an output file name.
//
static char
    i_file[] = "input_file_name.sio",
    o_file[] = "output_file_name.sio";

//
// Main routine.
//
int main()
{

SIO_record
   *dummy;

unsigned int
    status;

//
// Set up the SIO verbosity.
//
SIO_streamManager::setVerbosity( SIO_ALL );
SIO_recordManager::setVerbosity( SIO_ALL );
SIO_blockManager::setVerbosity( SIO_ALL );

//
// Create the SIO streams and open them. The memory allocations are just
// guesses.  SIO will adjust them as necessary, but it's useful to give
// a guideline.  The output stream is set up to overwrite a pre-existing
// file.
//
i_stream = SIO_streamManager::add( "input",  64 * SIO_KBYTE );
i_stream->open( i_file, SIO_MODE_READ );

o_stream = SIO_streamManager::add( "output", 64 * SIO_KBYTE );
o_stream->open( o_file, SIO_MODE_WRITE_NEW );

//
// Instantiate foo, bar and baz giving their SIO block name as the last
// argument (foo, bar and baz are presumed to have inherited from SIO_block).
//
Foo     *foo     = new Foo( ..., "Foo" );
Bar     *bar     = new Bar( ..., "Bar" );
Baz     *baz     = new Baz( ..., "Baz" );

//
// Create an SIO record for the input and request that these records be
// unpacked.  Note that it is not necessary to attach blocks to the
// input records.  SIO will use any block unpackers it has available.
// Nor is it necessary to tell SIO if the data is compressed or not, SIO
// can work that out from the data stream itself.
//
```

```
i_record = SIO_recordManager::add( "raw_event" );
i_record->setUnpack( true );

//
// Create an SIO record for the output and request that all output records
// of this type be written in compressed format. In this case, the blocks
// must be explicitly attached to get them written out.
//
o_record = SIO_recordManager::add( “baz_event” );
o_record->setCompress( true );
o_record->connect( foo );
o_record->connect( bar );
o_record->connect( baz );

//
// Loop over input records, manipulate them and write out new versions.
// Note that the input file could contain other records other than raw_data,
// but because raw_data is the only record type for which unpacking has been
// requested, all other records will be ignored.
//
for(;;)
{
    status = i_stream->read( &dummy );
    if( !(status & 1) )
        leave;

    baz->do_something( foo, bar );

    status = o_stream->write( “baz_event” );
    if( !(status & 1) )
        leave;
}

//
// That’s all folks!
//
return 0;
}
```

5 Questions And Answers

There always seems to be a bunch of extraneous information left over at the end of a document like this (the alternative … that I don't know how to structure good documentation … is of course completely unthinkable!).

I thought I'd try a new technique and present this information as a series of questions and answers.

Q Can I mix different records on the same stream?

A. Yes.

Q Can I mix compressed and uncompressed records on the same stream?

A. Yes.

Q Something went wrong while I was reading back my `SIO block`. How should I report this to SIO?

A. Instead of returning `SIO_BLOCK_SUCCESS`, return `SIO_BLOCK_NOTFOUND`. SIO will abort reading the current record and the condition code returned by `SIO_record::read` will reflect the failure. Reading will pick up again on the next read from the same stream.

Q Something went wrong while I was writing out my `SIO block`. How should I report this to SIO?

A. Instead of returning `SIO_BLOCK_SUCCESS`, return `SIO_BLOCK_NOTFOUND`. SIO will not write the current record and the condition code returned by

`SIO_record::write` will reflect the failure. Writing will pick up again on the next write to the same stream.

Q A call to `SIO_stream::getState` shows the stream to be in state `SIO_STATE_ERROR`. What can I do with the stream now?

A. Unfortunately not much. The only thing you can do with an errored stream is close it. If it closes successfully, you can reuse it. If anyone is interested in playing with SIO, this is an area that could certainly be improved.

Q How can I attach more than one block of the same name to a record?

A. You can't. Blocks are *not* objects. Think of a block as a data container or manager. It is free to write out multiple objects of the same or different types (and such objects could given an transfer method to be called by the block), or it can form its own data structures by summarizing the data in other objects. This is actually the start of an interesting theoretical discussion. There are ways SIO could be modified to make it a genuine C++ object reader/writer but these techniques quickly increase the complexity and were outside of the scope of the 'simple' binary format I was after.