User Manual

English

SIE and libsie
SoMat eDAQ/eDAQ-lite

I2954-1.0 en SoMat P/N DOC 0024-00

Safety Information

Conversions and modifications
HBM’s express consent is required for modifications affecting the SoMat eDAQ or eDAQ-lite design and safety. HBM does not take responsibility for damage resulting from unauthorized modifications.

Qualified personnel
The SIE format and libsie library may be used by qualified personnel only; the specifications and the special safety regulations need to be followed in all cases. The term “qualified personnel” refers to users with a reasonable background in software programming, particularly in C.

Terms of Use
The libsie library is free software and may be redistributed and/or modified under the terms of version 2.1 of the GNU Lesser General Public License as published by the Free Software Foundation. This document describes libsie version 1.0.0 and SIE version 1.0. All specifications are subject to change without notice in future version releases. The most recent version of this document can be found at www.hbm.com/somat.

Contents

1 Getting Started

1.1 Introduction

1.1.1 SIE Format

SIE is a new data transmission and storage format for engineering data. SIE is designed to be flexible, self-describing, streamable and robust, overcoming some of the limitations of existing engineering data formats. This is accomplished with a simple block structure and an XML description of how to read both the block structure and contained binary data making it very simple to write. SIE has been successfully implemented on the HBM SoMat eDAQ line of data acquisition systems and a library for reading SIE files (libsie) has been implemented in portable C. For detailed information on the SIE file, see “The SIE File” on page 31.

1.1.2 libsie Library

The libsie SIE reader library exposes an object-oriented C API for reading SIE data. libsie provides functions to open and traverse a file and to get the SIE data in a universal format. “Using the libsie Library” on page 9 provides a tutorial program to demonstrate most of the API. “libsie Library Reference” on page 19 contains a reference of the available libsie functions.

1.2 Required Downloads

Successful implementation of the libsie library requires the libsie software package and a C compiler.

1.2.1 libsie Library Download

The libsie package can be found at www.hbm.com/somat in the resource center. Note that registration is required for access. The libsie package include the libsie library files and a libsie-demo.c program on which the Chapter 2 tutorial of this manual is based. The libsie source is also available to allow the option of compiling it from scratch or using it on a platform other than Microsoft Windows®.

1.2.2 C Compiler Download

For those without an existing C compiler, the Microsoft® Visual C++ development environment is recommended. The free express version of the software is available for download at http://www.microsoft.com/express/vc/.

2 Using the libsie Library

This section describes the use of the libsie library through tutorial code that reads all data and metadata from an SIE file while demonstrating most of the libsie API. The code as presented is more linear than usually written to provide a structure better suited for the tutorial.

NOTE
This chapter is based on the libsie-demo.c program included in the libsie distribution.

2.1 Header and Main Function

First, include the necessary header files.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <sie.h>
```

Then, prototype the functions for the program. Both the `print_sie_file()` and the `print_tag()` functions are defined in this later in this program. For their definitions, see "Print SIE File Function Definition" on page 9 and "Print Tag Function Definition" on page 16.

```
static void print_sie_file(const char *filename);
static void print_tag(sie_Tag *tag, const char *prefix);
```

The `main()` function calls `print_sie_file()`.

```
int main(int argc, char **argv)
{
  if (argc < 2) {
    fprintf(stderr, "Please enter an SIE filename on "
          "the command line.\n");
    return EXIT_FAILURE;
  }

  print_sie_file(argv[1]);

  return EXIT_SUCCESS;
}
```

2.2 Print SIE File Function Definition

The `print_sie_file()` function prints the entire contents of an SIE file to standard output. The first step is to initialize the necessary variables.

```
void print_sie_file(const char *filename)
{
  sie_Context *context;
  sie_File *file;
  sie_Exception *exception;
  sie_Iterator *tag_iterator;
  sie_Iterator *test_iterator;
  sie_Iterator *channel_iterator;
  sie_Iterator *dimension_iterator;
  sie_Tag *tag;
  sie_Test *test;
  sie_Channel *channel;
  sie_Dimension *dimension;
  sie_Spigot *spigot;
  sie_Output *output;
  sie_uint32 id;
  int leaked_objects;
```

2.2.1 Library Context

The first step in using libsie is to get a library context.

```
context = sie_context_new();
```

This is used to hold internal library information about such things as buffers and error handling. When a library context reference is required, either a direct pointer to the context or a pointer to any other object that has been spawned from the context can be used.

NOTE
libsie is thread safe as long as code referencing a context or objects created from that context is only running one thread at a time. Objects cannot be shared among contexts.

Insert an error notice in the event of failure.

```
if (!context) {
  fprintf(stderr, "Error: Failed to acquire context!\n");
  exit(EXIT_FAILURE);
}
```

If acquiring the context fails, the problem has occurred at a very low level.

NOTE
All functions in the SIE API accept `NULL` pointers where they normally expect an SIE object and simple return an error value (usually `NULL` if the function would normally return an object pointer) when it is called with `NULL`. This allows an error to cascade through valid code to be checked at the end of the process. Obviously, non-SIE object pointers must be checked for `NULL` before being dereferenced or otherwise used as is typical in C code.

Next, open the desired SIE file.

```
file = sie_file_open(context, filename);
```

Notice that the context is passed into `sie_file_open()`. The SIE file object that is returned references the same context and can be used to refer to it as an argument to other functions that need a context. For instance, to open another file, the following two statements are identical.

```
file2 = sie_file_open(context, filename2);

file2 = sie_file_open(file, filename2);
```

2.2.2 Error Handling

To demonstrate the libsie error-handling tools, insert a catch for a possible error, print the exception and then exit. An error for the `sie_file_open()` function may be caused by a non-existent or corrupted file.

```
if (!file) {
  exception = sie_get_exception(context);
  fprintf(stderr, "Something bad happened:\n  %s\n",
        sie_verbose_report(exception));

  exit(EXIT_FAILURE);
}
```

Functions that return SIE objects return `NULL` on failure. If `file` is `NULL`, the requested file failed to open. The function `sie_get_exception()` pulls the exception out of the library context. To get a string describing the exception, use either `sie_report()` or `sie_verbose_report()`. In addition to the exception message, the verbose version contains what the library was doing when the exception happened.
On a successful open operation, use the following command to view the file name.

```
printf("File '%s':\n", filename);
```

2.2.3 Iterators and File-Level Tags

The generic form of metadata in an SIE file is called a tag. This is simply a reference between an arbitrary textual key and an arbitrary value. Tags can exist at any level in the SIE metadata hierarchy, including at the file level.
libsie functions that return more than one object do so via an object called an iterator. The iterator returns the other objects one at a time when the `sie_iterator_next()` method is called. When the iterator is empty, `sie_iterator_next()` returns `NULL` instead of an object.
To print out the file-level tags in the file, get an iterator containing all of the object's tags using `sie_get_tags()`.

```
tag_iterator = sie_get_tags(file);
```

Next, pull all the tags out of the iterator, one at a time. The `print_tag()` function is defined in “Print Tag Function Definition” on page 16.

```
while ((tag = sie_iterator_next(tag_iterator))) {
  print_tag(tag, " File tag ");
}
```

After pulling out all the tags, release the iterator. This informs libsie that the iterator is no longer in use.

```
sie_release(tag_iterator);
```

it is not necessary to release the tag objects because the objects returned by an iterator are still owned by it. This allows for a shorter iterating pattern. If an iterator returned object is needed after getting the next object or releasing the iterator, simply call `sie_retain()` on the object. This is the opposite of `sie_release()`, informing libsie that the object is needed again and it will be released later.

NOTE
In terms of referencing counting memory management, `sie_retain()` raises the reference count and `sie_release()` lowers the reference count.

2.2.4 SIE Tests and Test-Level Tags

The next step is to get all the test runs contained in the SIE file. SIE tests are grouped collections of channels. Again, use an iterator to get the SIE tests and cycle through the iterator to return each test object. Within an SIE file, each test has a numeric ID which is a unique identifier for that test within the SIE file. To demonstrate, print the test ID for each test in the iterator.

```
test_iterator = sie_get_tests(file);
while ((test = sie_iterator_next(test_iterator))) {
  id = sie_get_id(test);
  printf("  Test id %lu:\n", (unsigned long)id);
```

Like the SIE file itself, tests also have tags. Print these just like the file-level tags in the previous section and release the tag iterator.

```
  tag_iterator = sie_get_tags(test);
  while ((tag = sie_iterator_next(tag_iterator)))
    print_tag(tag, "    Test tag ");
  sie_release(tag_iterator);
```

2.2.5 Channels and Channel-Level Tags

Finally, tests contain the channels that were collected as a part of the test run. To get these channels, follow the same pattern used previously. Note that channels have an SIE-internal ID just like tests. They also have a name, accessible with the function `sie_get_name()`.

```
  channel_iterator = sie_get_channels(test);
  while ((channel = sie_iterator_next(channel_iterator))) {
    printf("    Channel id %lu, '%s':\n",
          (unsigned long)sie_get_id(channel),
          sie_get_name(channel));
```

Channels also contain tags. Although presented inline in this tutorial code, in an actual program, it is probably more desirable to use a function to print groups of tags.

```
tag_iterator = sie_get_tags(channel);
while ((tag = sie_iterator_next(tag_iterator)))
  print_tag(tag, "      Channel tag ");
sie_release(tag_iterator);
```

Getting Channels Directly From a File
It is possible to skip the test level of the hierarchy and get all the channels directly in a file. For example:

```
channel_iterator = sie_get_channels(file);
while ((channel = sie_iterator_next(channel_iterator))) {
  printf("  Channel id %lu, '%s' ",
        (unsigned long)sie_get_id(channel), sie_get_name(channel));
```

Use `sie_get_containing_test()` to return the test that contains a channel. Note that not all channels must be contained by a test, though most containing actual user data will.

```
  test = sie_get_containing_test(channel);
  if (test)
    printf("is contained in test id %lu.\n",
          (unsigned long)sie_get_id(test));
  else
    printf("is not in a test.\n");
  sie_release(test);
}
sie_release(channel_iterator);
```

2.2.6 Channel Dimensions

Channels contain dimensions which define an axis or column of data. Each dimension has an index. For example, in a typical time series, dimension index 0 is time and dimension index 1 is the engineering value of the data. The dimensions also have tags.

```
dimension_iterator = sie_get_dimensions(channel);
while ((dimension = sie_iterator_next(dimension_iterator))) {
  printf("      Dimension index %lu:\n",
        (unsigned long)sie_get_index(dimension));

  tag_iterator = sie_get_tags(dimension);
  while ((tag = sie_iterator_next(tag_iterator)))
    print_tag(tag, "        Dimension tag ");
  sie_release(tag_iterator);
}
sie_release(dimension_iterator);
```

2.2.7 Spigots

Finally, channels can have a spigot attached to them to get the data. libsie presents data as an array or matrix where each column is a dimension, as defined in the previous section. Each column can either consist of 64-bit floats or of raw octet strings. For example, a time series may be represented as follows:

Dimension 0	Dimension 1
0.0	0.0
1.0	0.25
2.0	0.5
3.0	0.25
4.0	0.0

NOTE
While all data comes out in this general form, there are multiple ways to interpret the output. To see which type of data is stored and how the channel output should be interpreted, look at the channel tag `core:schema`. For the example above, if the time series was generated by an eDAQ, an appropriate schema would be `somat:sequential`. In this schema, dimension 0 is time and dimension 1 is the data value of the channel. All numbers come out scaled to their engineering values. For more information on SoMat data schemas, see "SoMat Data Schema" on page 46.

To read the data stored in the channel, first attach a spigot. As with iterators, a spigot contains a sequential sections of the channel's data. The data is arranged into blocks in the SIE file, and cycling through the spigot gets one block at a time. The data comes out in an `sie_Output` object and the spigot is read with `sie_spigot_get()`.

```
spigot = sie_attach_spigot(channel);
while ((output = sie_spigot_get(spigot))) {
  sie_Output_Struct *os;
  size_t dim, row, num_dims, num_rows, byte, size;
  unsigned char *uchar_p;
```

To read the number of dimensions (i.e., columns) and the number of rows, use the `sie_output_get_num_dims()` and `sie_output_get_num_rows()`, respectively.

```
num_dims = sie_output_get_num_dims(output);
num_rows = sie_output_get_num_rows(output);
printf("     Data block %lu, %lu dimensions, %lu rows:\n",
       (unsigned long)sie_output_get_block(output),
       (unsigned long)num_dims, (unsigned long)num_rows);
```

libsie offers several methods to get the data out of the output object. One method, more suitable for languages that don't have easy access to C structs, is to use `sie_output_get_float64()` or `sie_output_get_raw()` to retrieve an array containing one dimension's data. Use `sie_output_get_type()` to return the type of data in the dimension.
However, in languages that can interpret C structs, it is possible to pull out one struct that contains all of the data.

```
os = sie_output_get_struct(output);
```

NOTE
A struct is not an SIE object, so `sie_release()` cannot be used on it. Rather, it is owned by the output object and is released with that object.

Next, iterate through the C struct, printing all the data. The type can be `SIE_OUTPUT_FLOAT64` or `SIE_OUTPUT_RAW`. Raw data has three parts: a pointer to the data (`ptr`), the size of the data (`size`) and a `claimed` field. Set `claimed` to 1 to keep the data and clean up the associated memory manually. Otherwise, the raw data is cleaned up with the rest of the output when the output object is released.

```
for (row = 0; row < num_rows; row++) {
  printf("        Row %lu: ", (unsigned long)row);
  for (dim = 0; dim < num_dims; dim++) {
    if (dim != 0)
      printf(", ");
    switch (os->dim[dim].type) {
    case SIE_OUTPUT_FLOAT64:
      printf("%.15g", os->dim[dim].float64[row]);
      break;
    case SIE_OUTPUT_RAW:
      uchar_p = os->dim[dim].raw[row].ptr;
      size = os->dim[dim].raw[row].size;
      if (size > 16) {
        printf("(raw data of size %lu.)", (unsigned long)size);
      } else {
        for (byte = 0; byte < size; byte++)
          printf("%02x", uchar_p[byte]);
      }
      break;
    }
  }
  printf("\n");
}
```

As with iterators, there is no need to release the output object as the spigot still owns it and it is possible to retain the object if necessary. To begin closing out the function, release the spigot, iterators and file. Also, check for any exceptions that may have occurred during the run.

```
      }
      sie_release(spigot);
    }
    sie_release(channel_iterator);
  }
  sie_release(test_iterator);
  sie_release(file);
  if (sie_check_exception(context)) {
    exception = sie_get_exception(context);
    fprintf(stderr, "Something bad happened:\n %s\n",
          sie_verbose_report(exception));
    sie_release(exception);
  }
```

2.2.8 Releasing a Context

Unlike other SIE objects, a context must be disposed of in a special way to release internal data structures and break circular references. The `sie_context_done()` function attempts to dispose of the specified context. If successful, it returns zero. Otherwise, it returns the number of other objects still alive and referencing the context. It is good practice to check that this returns zero to ensure all SIE objects are cleaned up properly.

```
  leaked_objects = sie_context_done(context);
  if (leaked_objects != 0)
    fprintf(stderr, "Warning: Leaked %d SIE objects!\n",
          leaked_objects);
}
```

NOTE

Unlike other functions that need a context as an argument, the actual context object must be used when calling `sie_context_done()`. Using another live object to refer to the context would cause a failure as that object, by definition, would still be referencing the context.

2.3 Print Tag Function Definition

The `print_tag()` function prints a tag to standard output, prefixed by a string. It is used many times in the previous `print_sie_file()` function. First, initialize the necessary variables.

```
static void print_tag(sie_Tag *tag, const char *prefix)
{
  const char *name;
  char *value = NULL;
  size_t value_size = 0;
  int dont_print = 0;
```

As mentioned previously, a tag is a relation between a textual key (i.e., ID) and an arbitrary value. Getting the ID is simple:

```
  name = sie_tag_get_id(tag);
```

Tags can contain arbitrary-length binary data in the value. To get the entire contents of the tag value in a binary-safe way, use `sie_tag_get_value_b()`.

```
  sie_tag_get_value_b(tag, &value, &value_size);
```

This sets `value` as a pointer to newly allocated memory of size `value_size` containing the tag `value`. The value is guaranteed to be `NULL`-terminated by the library, so it can be treated safely as a C string. However, the `NULL` added by libsie is not included in the size returned so as to not disturb the size information of the real binary data. If successful, `sie_tag_get_value_b()` returns true and, if not, it returns zero.

NOTE
Because a tag value can be arbitrarily long, allocating its entirety to memory (as the above function does) may not be wise. To avoid this, attach a spigot to the tag to get the value out in sections just as a spigot is used to get data out of a channel.

As tags are occasionally long, this example function prints only the length of the tag if the tag value is over 50 bytes or contains any nulls.

```
    if (value_size > 50 || memchr(value, 0, value_size)) {
      printf("%s'%s': long tag of %lu bytes.\n", prefix, name,
            (unsigned long)value_size);
      dont_print = 1;
    }
```

The value returned by `sie_tag_get_value_b()` must be freed as any other allocated raw memory in C. To free plain pointers to allocated memory returned from libsie. call `sie_free()`.

```
    sie_free(value);

    if (dont_print)
      return;
```

If it is necessary to know the binary size of the tag value, the value returned by `sie_tag_get_value_b()` can be printed. If, however, the binary size is not needed, the tag value can be treated as a `NULL`-terminated string using the `sie_tag_get_value()` function. Again, the returned value must be freed.

```
    value = sie_tag_get_value(tag);

    printf("%s'%s': '%s'\n", prefix, name, value);

    sie_free(value);

  }
```

NOTE
The ID returned by `sie_tag_get_id()` does not need to be freed. It is cleaned up with the tag object.

3 libsie Library Reference

3.1 Overview

The SIE reader library (libsie) exposes an object-oriented C API for reading SIE data. It provides functions to open and traverse an SIE file and get its data in a universal format. This chapter details the functions available in the libsie library.

Please note the following:

- All functions are prefixed with `sie_`.
- All of the `sie_*` object pointer types can be considered generic pointers -- a normal library user should not have need to access the internals.
- The special types `sie_float64` and `sie_uint32` are used in these definitions. Their C equivalents are architecture-specific.

NOTE

This document describes libsie version 1.0.0.

3.2 Memory Management

libsie uses reference counting for its memory handling. There are two generic methods to manage an object's reference count: retain and release.

`sie_retain()` **Retain**

Raise an object's reference count by one.

```
void *sie_retain(void *object);
```

A `NULL` input is safe and does nothing

`sie_release()` **Release**

Lower an object's reference count by one. If an object's count reaches zero, it is freed from memory.

```
void sie_release(void *object);
```

A `NULL` input is safe and does nothing. The `sie_retain()` method returns the object pointer allowing for chaining constructs like `return sie_retain(object)`.

`sie_free()` **SIE Free**

Free the libsie-allocated memory pointed to by the pointer.

```
void sie_free(void *pointer);
```

3.3 Library Context

The first step towards using the library is to get a library context. This serves to keep global resources around that are used in library operations. It is also used for error handling. libsie is thread-safe as long as code referencing a context is only running in a single thread at a time. Multiple contexts can be created, but objects cannot be shared between them.

`sie_context_new()`

New Context
Create a new library context.

```
sie_Context *sie_context_new(void);
```

Other API functions take a context object that specifies in what library context the function should operate. Any libsie object, including the context itself, can function as a context object referring to the context in which it was created.

`sie_context_done()`

Release a Context
Context objects are an exception to the generic retain and release object methods. To release a library context, use the following method:

```
int sie_context_done(sie_Context *context);
```

If successful, the function returns zero. Otherwise, the function returns the number of objects that still have dangling references to them and, as such, were not freed.

3.4 Iterators

An iterator allows access to a collection of objects, such as channels or tags.

`sie_iterator_next()`

Return Object
Return the next object in an iterator.

```
void *sie_iterator_next(void *iterator);
```

The returned object, owned by the spigot, is valid until the next call to `sie_iterator_next()` and does not need to be released. To reference the object after the next call to `sie_iterator_next()` or the release of the iterator, use `sie_retain()` to retain the object and `sie_release()` to release it later.

3.5 Spigots

A spigot is the interface used to get data out of the library. A spigot can be attached to several kinds of references (currently, channels and tags) and can be read from repeatedly, returning the data contained in the reference.
The data that comes out of a spigot is arranged in scans of vectors. Each column can be one of two data types: 64-bit float or raw, which is a string of octets. The form of the data varies depending on the channel.

`sie_attach_spigot()`

Attach a Spigot
Attach a spigot to a reference in preparation for reading data.

```
sie_Spigot *sie_attach_spigot(void *reference);
```

`sie_spigot_get()`

Get Output
Read the next output record out of spigot. If it returns `NULL`, all data has been read.

```
sie_Output *sie_spigot_get(sie_Spigot *spigot);
```

The output record, owned by the spigot, is valid until the next call to `sie_spigot_get()` and does not need to be released. To reference the output after the next call to `sie_spigot_get()` or the release of the spigot, use `sie_retain()` to retain the object and `sie_release()` to release it later.

`sie_spigot_disable_transforms()`

Disable Transformations
Disable data transformations by SIE `xform` nodes.

```
void sie_spigot_disable_transforms(void *spigot, int disable);
```

If `disable` is true, the data returned is not transformed. This typically means that raw decoder output is returned instead of engineering values. This can be useful as many data schemas have dimension 0 as time when scaled and as sample count when unscaled. When using unscaled data and binary search, dimension 0 can be used to find a particular sample number with such schemas. This is currently applicable only to channels.

`sie_spigot_transform_output()`

Transform Output
Transform the output as it would be had transforms not been disabled.

```
void sie_spigot_transform_output(void *spigot, sie_Output *output);
```

This method allows transforming the spigot output after the fact to get both the transformed and non-transformed outputs without reading from the spigot twice.

`sie_spigot_seek()`

Seek
Prepare the spigot such that the next call to `sie_spigot_get()` returns the data in the block target.

```
#define SIE_SPIGOT_SEEK_END(~(size_t)0)
size_t sie_spigot_seek(void *spigot, size_t target);
```

If the specified target is past the end of the data in the file, the method sets it to the end of the data. For example, setting the target to one block after the last one causes the next call of `sie_spigot_get()` to return `NULL` indicating the end of the data. The method returns the block position that was set. `SIE_SPIGOT_SEEK_END`, defined as all ones (i.e., 0xffffffff on 32-bit platforms), is provided as a convenient way to seek to the end of a file.

`sie_spigot_tell()`

Get Current Position
Return the current block position of the spigot which is the block that the next call to `sie_spigot_get()` returns.

```
size_t sie_spigot_tell(void *spigot);
```

`sie_lower_bound()`

Lower Bound Search
Find the block of data where the value of a specified dimension is first greater than or equal to a specified value.

```
int sie_lower_bound(void *spigot, size_t dim,
      sie_float64 value, size_t *block, size_t *scan);
```

The dimension specified by `dim` must be non-decreasing. If a value greater than or equal to `value` is found, the function scans within the found block, returns true and sects `block` and `scan` to the found value. The parameters `block` and `scan` are always returned such that seeking to `block`, calling `sie_spigot_get()` and getting the scan number `scan` in that block returns the first value in the dimension greater than or equal to the search value. The function returns false if the last point in the data is less than the value or if some other error ocurred. The block the spigot is currently pointing at is not affected.

`sie_upper_bound()`

Upper Bound Search
Find the block of data where the value of a specified dimension is first less than or equal to a specified value.

```
int sie_upper_bound(void *spigot, size_t dim,
      sie_float64 value, size_t *block, size_t *scan);
```

The dimension specified by `dim` must be non-decreasing. If a value less than or equal to `value` is found, the function scans within the found block, returns true and sects `block` and `scan` to the found value. The parameters `block` and `scan` are always returned such that seeking to `block`, calling `sie_spigot_get()` and getting the scan number `scan` in that block returns the first value in the dimension less than or equal to the search value. The function returns false if the last point in the data is greater than the value or if some other error ocurred. The block the spigot is currently pointing at is not affected.

`sie_spigot_done()`

Check if Spigot is Done
Return true if the specified spigot is done, meaning that all data have been read and no more data will ever appear on the spigot.

```
int sie_spigot_done(void *spigot);
```

3.6 Reference Methods

References include files, tests, channels, dimensions and tags. The methods available for each type of reference are detailed below. A reference method used for an invalid type of reference returns `NULL` unless otherwise specified.

3.6.1 Files

`sie_file_is_sie()`

SIE File Test
Quickly test if the file specified by `name` looks like an SIE file. Returns non-zero if it looks like an SIE file or zero otherwise.

```
int sie_file_is_sie(void *ctx_obj, const char *name);
```

`sie_file_open()`

Open SIE File
Open an SIE file, returning a file object.

```
sie_File *sie_file_open(void *context_object, const char *name);
```

`sie_ignore_trailing_garbage()`

Ignore Trailing Garbage
Open a corrupted file with a valid block at the end of the file of a specified size.

```
void sie_ignore_trailing_garbage(void *ctx_obj, size_t amount);
```

By default, the library refuses to open SIE files with any detectable corruption. The `sie_file_open()` method returns `NULL` and sets the library exception to an explanation of the error. However, a somewhat common corruption is a file truncated in the middle of a block which can occur when reading a file that is being written at the same time. The `sie_ignore_trailing_garbage()` method tells the library to open the file anyway, as long as it finds a valid block at the end of the file of the size specified by `amount`.

`sie_get_tests()`

Get All Tests
Return an iterator containing all tests in the file as `sie_Test` objects.

```
sie_Iterator *sie_get_tests(void *reference);
```

`sie_get_channels()`

Get All Channels
Return an iterator containing all channels (as `sie_Channel` objects) in the file.

```
sie_Iterator *sie_get_channels(void *reference);
```

`sie_get_tags()`

Get All Tags
Return an iterator containing all top level tags in the file.

```
sie_Iterator *sie_get_tags(void *reference);
```

`sie_get_test()`

Get Test
Return the test in the file with its ID or `NULL` if no such test ID exists.

```
sie_Test *sie_get_test(void *reference, sie_uint32 id);
```

`sie_get_channel()`

Get Channel
Return the channel in the test with its ID or `NULL` if no such channel ID exists.

```
sie_Channel *sie_get_channel(void *reference, sie_uint32 id);
```

`sie_get_tag()`

Get Tag
Return the tag in the file with its ID or `NULL` if no such tag ID exists.

```
sie_Tag *sie_get_tag(void *reference, const char *id);
```

3.6.2 Tests

`sie_get_channels()`

Get All Channels
Return an iterator containing all channels (as `sie_Channel` objects) in the test.

```
sie_Iterator *sie_get_channels(void *reference);
```

`sie_get_tags()`

Get All Tags
Return an iterator containing all tags in the test.

```
sie_Iterator *sie_get_tags(void *reference);
```

`sie_get_tag()`

Get Tag
Return the tag in the test with its ID or `NULL` if no such tag ID exists.

```
sie_Tag *sie_get_tag(void *reference, const char *id);
```

`sie_get_id()`

Get Test ID
Return the ID of the test. The method returns `SIE_NULL_ID` if the call is invalid.

```
#define SIE_NULL_ID   (~(sie_uint32)0)
sie_uint32 sie_get_id(void *reference);
```

3.6.3 Channels

`sie_get_dimensions()`

Get All Dimensions
Return an iterator containing all dimensions (as `sie_Deminsion` objects) in the channel.

```
sie_Iterator *sie_get_dimensions(void *reference);
```

`sie_get_tags()` **Get All Tags**
Return an iterator containing all tags in the channel.

```
sie_Iterator *sie_get_tags(void *reference);
```

`sie_get_dimension()` **Get Dimension**
Return the dimension in the channel with its index or `NULL` if no such dimension exists.

```
sie_Dimension *sie_get_dimension(void *reference, sie_uint32 index);
```

`sie_get_tag()` **Get Tag**
Return the tag in the channel with its ID or `NULL` if no such tag ID exists.

```
sie_Tag *sie_get_tag(void *reference, const char *id);
```

`sie_get_containing_test()` **Get Containing Test**
Return the test the channel is a member of, if any.

```
sie_Test *sie_get_containing_test(void *reference);
```

`sie_get_name()` **Get Channel Name**
Return the name of the channel.

```
const char *sie_get_name(void *reference);
```

`sie_get_id()` **Get Channel ID**
Return the ID of the channel. The method returns `SIE_NULL_ID` if the call is invalid.

```
#define SIE_NULL_ID   (~(sie_uint32)0)
sie_uint32 sie_get_id(void *reference);
```

3.6.4 Dimensions

`sie_get_tags()` **Get All Tags**
Return all tags in the dimension.

```
sie_Iterator *sie_get_tags(void *reference);
```

`sie_get_tag()` **Get Tag**
Return the tag in dimension with its ID or `NULL` if no such tag ID exists.

```
sie_Tag *sie_get_tag(void *reference, const char *id);
```

`sie_get_index()` **Get Dimension Index**
Return the index of the dimension.

```
sie_uint32 sie_get_index(void *reference);
```

3.6.5 Tags

A tag is a key to value pairing and is used for almost all metadata. Use the following methods to access tags.

`sie_tag_get_id()` **Get Tag ID**
Return the ID of a tag.

```
const char *sie_tag_get_id(sie_Tag *tag);
```

The returned string is valid for the lifetime of the tag object. If needed longer, duplicate the string using the `strdup` function or otherwise reallocate the string.

`sie_tag_get_value()`

Get Tag Value
Return a newly-allocated string containing the value of a tag.

```
char *sie_tag_get_value(sie_Tag *tag);
```

This function returns `NULL` on failure. Because the returned string is a plain pointer, it must eventually be freed with the `sie_free` function.

NOTE
Because the amount of data in a tag value can be very large, it can also be read with a spigot. For more information, see "Spigots" on page 20.

`sie_tag_get_value_b()`

Get Tag Value (Binary Safe)
Set a pointer to a newly-allocated string containing the tag value and a pointer to the size of the data.

```
int sie_tag_get_value_b(sie_Tag *tag, char **value, size_t *size);
```

The pointer `value` points to the value of `tag` and the pointer `size` points to the size of the data. This function returns true if successful. If an error occurs, the function returns false and `value` and `size` are unchanged. If successful, value must eventually be freed with the `sie_free()` function.

NOTE
Because the amount of data in a tag value can be very large, it can also be read with a spigot. For more information, see "Spigots" on page 20.

3.7 Output Methods

Use the following methods to access the output data from a spigot. The data from a spigot is arranged in scans of vectors. The data type of each column can be either 64-bit float or raw. The form of the data varies depending on the channel.

`sie_output_get_block()`

Get Block Number
Return the number of the block from which the data originated. The first data block in a channel is always block 0.

```
size_t sie_output_get_block(sie_Output *output);
```

`sie_output_get_num_dims()`

Get Number of Dimensions
Return the number of dimensions in the output.

```
size_t sie_output_get_num_dims(sie_Output *output);
```

`sie_output_get_num_rows()`

Get Number of Rows
Return the number of rows of data in the output.

```
size_t sie_output_get_num_rows(sie_Output *output);
```

`sie_output_get_type()`

Get Data Type
Return the type of the specified dimension of the output.

```
#define SIE_OUTPUT_NONE    0
#define SIE_OUTPUT_FLOAT64 1
#define SIE_OUTPUT_RAW     2
int sie_output_get_type(sie_Output *output, size_t dim);
```

sie_output_get_float64()

Get Float Data
Return a pointer to an array of float64 (i.e., double) data for the specified dimension.

```
sie_float64 *sie_output_get_float64(sie_Output *output, size_t dim);
```

The data array has a size equal to the number of scans in the output. This is only valid if the type of the dimension is `SIE_OUTPUT_FLOAT64`. The lifetime of the return value is managed by the `sie_Output` object.

sie_output_get_raw()

Get Raw Data
Return a pointer to an array of raw data for the specified dimension.

```
typedef struct _sie_Output_Raw {
  void *ptr;
  size_t size;
  int reserved_1;
} sie_Output_Raw;
```

```
sie_Output_Raw *sie_output_get_raw(sie_Output *output, size_t dim);
```

The data array has a size equal to the number of scans in the output. The `ptr` member of the `sie_Output_Raw` struct is a pointer to the actual data and `size` is the size in bytes of the data pointed to by the pointer. The lifetime of the return value is managed by the `sie_Output` object.

sie_output_get_struct()

Get Struct
Return an `sie_Output_Struct` pointer containing information about the `sie_Output` object.

```
typedef struct _sie_Output_Dim {
  int type;
  sie_float64 *float64;
  sie_Output_Raw *raw;
} sie_Output_Dim;
```

```
typedef struct _sie_Output_Struct {
  size_t num_dims;
  size_t num_rows;
  size_t reserved_1;
  size_t reserved_2;
  sie_Output_Dim *dim;
} sie_Output_Struct;
```

```
sie_Output_Struct *sie_output_get_struct(sie_Output *output);
```

The returned struct can be used in all C-compatible languages to access all data in the `sie_Output` object. The lifetime of the return value is managed by the `sie_Output` object.
For information about the SIE data model, see “The SIE File” on page 31.

3.8 Error Handling

`sie_check_exception()`

Check for Exception
Check for an exception since library initialization or the last call to `sie_get_exception()`.

```
sie_Exception *sie_check_exception(void *ctx_obj);
```

The method returns `NULL` if no exception has occurred. Otherwise, it returns a non-`NULL` pointer.

`sie_get_exception()`

Get Exception
Return the exception object.

```
sie_Exception *sie_get_exception(void *ctx_obj);
```

The method returns `NULL` if no exception has occurred since library initialization or the last call to `sie_get_exception()`. The caller is responsible for releasing the exception object when finished.

`sie_report()`

Get Exception Report
Return a string describing the exception.

```
char *sie_report(void *exception);
```

The returned string is valid for the lifetime of the exception object and does not have to be freed.

`sie_verbose_report()`

Get Verbose Exception Report
Return a string describing the exception plus extra information describing what was happening when the exception occurred.

```
char *sie_verbose_report(void *exception);
```

The returned string is valid for the lifetime of the exception object and does not have to be freed.

3.9 Progress Information

Using some operations, such as sie_file_open(), on very large SIE files can take enough time that progress information may be useful. The following interface allows configuration of the SIE library context to provide information on the progress of libsie activities.
If a callback returns non-zero, the current API function is aborted. The API function returns a failure value and an operation aborted exception.

```
typedef int (sie_Progress_Set_Message)(void *data,
      const char *message);
typedef int (sie_Progress_Percent)(void *data,
      sie_float64 percent_done);

void sie_set_progress_callbacks(void *ctx_obj, void *data,
      sie_Progress_Set_Message *set_message_callback,
      sie_Progress_Percent *percent_callback);
```

3.10 Streaming

See contrib/misc/stream-test.c for an example of how to use streaming.

`sie_stream_new()`

Create SIE Stream
Create a new SIE stream.

```
sie_Stream *sie_stream_new(void *context_object);
```

The stream object can be used in all places a file object can be used. Data from the stream can only be read once.

`sie_add_stream_data()`

Add Stream Data
Add data to an existing stream.

```
size_t sie_add_stream_data(void *stream, const void *data,
      size_t size);
```

The method adds `size` bytes pointed to by the pointer `data` to the SIE stream `stream`. The function returns `size` if successful or zero if the stream is corrupt. After the function returns, query open spigots for more data or open new channels.

3.11 Histogram Access

Histograms are presented with a data schema which is comprehensive but somewhat inconvenient to access. However, libsie provides a utility for reconstructing a more traditional data representation.
The SoMat histogram data schema for each bin is:

```
dim 0: count
dim 1: dimension 0 lower bound
dim 2: dimension 0 upper bound
dim 3: dimension 1 lower bound
dim 4: dimension 1 upper bound
...
```

for as many dimensions as are present in the histogram. If a bin is repeated, the new count replaces the old one. This presents all the data needed to reconstruct the histogram in one place.
In SoMat files, this schema is used whenever the core schema tag (`core:schema`) is `somat:histogram` or `somat:rainflow`. When using rainflow data, this schema is used with an additional tag of rainflow stack data. For more information on these SoMat data schemas, see "Histogram Data Schema" on page 47.
To access a histogram in a more convenient way, use the following methods to create and interface with histogram objects.

`sie_histogram_new()`

Create a New Histogram
Create a new histogram convenience object from the specified channel and read all data from the channel.

```
sie_Histogram *sie_histogram_new(sie_Channel *channel);
```

`sie_histogram_get_num_dims()`

Get the Number of Dimensions
Return the number of dimensions in the histogram.

```
size_t sie_histogram_get_num_dims(sie_Histogram *hist);
```

`sie_histogram_get_num_bins()`

Get the Number of Bins
Return the number of bins in the specified dimension of the histogram.

```
size_t sie_histogram_get_num_bins(sie_Histogram *hist, size_t dim);
```

sie_histogram_get_bin_bounds()

Get Bin Bounds
Get the lower and upper bounds of the bins in the specified dimension.

```
void sie_histogram_get_bin_bounds(sie_Histogram *hist,
      size_t dim, sie_float64 *lower, sie_float64 *upper);
```

The method fills the arrays `lower` and `upper` with the lower and upper bounds of the bins. The arrays must have enough space for the number of bins in the dimension (see the `sie_histogram_get_num_bins()` function above).

sie_histogram_get_bin()

Get Bin
Return the bin value for the specified indices.

```
sie_float64 sie_histogram_get_bin(sie_Histogram *hist,
      size_t *indices);
```

The parameter `indices` must point to an array of size `size_t`. The size is defined as the number of dimensions of the histogram.

sie_histogram_get_next_nonzero_bin()

Get Next Non-Zero Bin
Return the bin value of the next non-zero bin from a specified starting position.

```
sie_float64 sie_histogram_get_next_nonzero_bin(
      sie_Histogram *hist, size_t *start, size_t *indices);
```

The method sets the parameter `indicies` to the indices of the found bin and the parameter `start` to a value that allows future calls of the function to continue the search from the bin after the current found bin. To start a new search, set `start` to point to a zero value. The parameter `indices` must point to an array of size `size_t`. The size is defined as the number of dimensions of the histogram. When there are no more non-zero bins, the function returns 0.0.

4 The SIE File

This appendix presents the details of the SIE file itself including the format of the SIE file, the structure and behavior of the underlying XML and the core and SoMat SIE schemas.

4.1 Overview

SIE is a flexible and robust data storage and transmission format and it is designed to be self-describing. At its core, SIE is a block-structured container for data. It defines an XML schema for describing both the structure and associated metadata of the data. It also defines an optional index format to speed up file access.

NOTE
This document describes SIE version 1.0.

4.2 SIE Format

4.2.1 Data Model

SIE presents a flexible, unified data model to the end user. Data are presented as a table of rows and columns. For example, a time series channel could be represented as the following:

Row	Dimension 0	Dimension 1
0	0.0	0.42
1	0.1	-0.20
2	0.2	0.13
3	0.3	0.06
4	0.4	-0.23
...	...	...

A dimension (column) may contain either numerical values or raw binary data.
The metadata for this example contains additional information, such as that dimension 0 is time with the unit seconds, while dimension 1 is the actual data measurement with the unit millivolts.
Ultimately, all SIE data are presented in this basic form. It is notable that, if given an SIE file with completely unknown data, all of the data values are immediately visible even if the details of the intended representation are unknown.
Metadata are represented as tags, which are pairings of a textual name to arbitrary binary data and can exist an any level of the SIE hierarchy. The metadata for the example above could be represented as:

```
<ch id="0" name="example">
  <dim index="0">
    <tag id="core:label">time</tag>
```

```
      <tag id="core:units">seconds</tag>
    </dim>
    <dim index="1">
      <tag id="core:label">measurement</tag>
      <tag id="core:units">millivolts</tag>
    </dim>
  </ch>
```

To help standardize and document the many ways that data and metadata can be represented on this basic model, SIE provides schemas, which are documented sets of data and metadata representation standards. Schemas are identified by a namespacing system. For example, all tags beginning with `core:` belong to the core schema. The SIE schema, described in “libsie Library Reference” on page 19, is reserved for items defined in the SIE format proper. Two other schemas, the core schema and SoMat schema, are discussed in this appendix in “Core Schema” on page 42 and “SoMat Schema” on page 44.

Note that SIE can store data in almost any conceivable format. Most of the SIE format is in fact a mechanism to convert arbitrary binary data in a block-structured container to the unified data model described above.

4.2.2 Block Structure

SIE is output as a stream of bytes, which is composed of a series of data blocks in sequence without intervening padding as depicted in figure 1. A block is composed of several parts, namely a size, group, sync word, payload and checksum. The size is repeated on both ends of the block. All of these except the payload are 32-bit unsigned integers in network (big-endian) byte order.

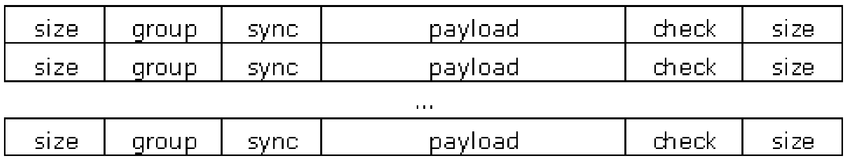
size | group | sync | payload | check | size
size | group | sync | payload | check | size
...
size | group | sync | payload | check | size

Figure 4-1: Figure 1: An SIE data stream

The size is simply the size of the block in bytes, including both size fields. Having the size on both sides of the block allows an SIE data file to be traversed both forwards and backwards and it provides one level of consistency checking in that the size fields in each block should always be equal.

The group identifies what data are contained in the payload of the block. Groups 0 and 1 indicate the XML metadata and index blocks, respectively. The purposes of all other groups are defined in the XML metadata.

The sync word is always the hexadecimal value `51EDA7A0`, which looks vaguely like “SIEDATA0.” The sync word is used to find the beginning of a block when reading damaged files or joining a broadcast stream in the middle. Also, as it has some high bits set, if an SIE file is transmitted over a medium that is not 8-bit clean, the sync word will be damaged, quickly indicating corruption even if checksums are not present.

The payload is the actual data contained in the block. It is interpreted based on the group of the block. A block with a payload size of zero indicates that no more blocks of that group will occur. This is intended to be used to indicate end of data in real-time streaming scenarios. The length of the payload is always the block size minus 20. Finally, the checksum is a CRC32 checksum of all bytes from the first byte of the first size field through the last byte of the payload. The checksum is optional, as some devices may not have the processor capacity to compute it, but, if it is not present, the checksum field must be set to zero. When reading a block, a zero checksum is always valid.

Offset (bytes)	Size (bytes)	Name	Description
0	4	size	Total size of the block in bytes.
4	4	group	Group identifier for the block.
8	4	sync word	Constant `51EDA7A0`.
12	n	payload	n bytes of data
n+12	4	checksum	CRC32 of bytes 0 through n+11 inclusive.
n+16	4	size2	Total size of the block in bytes.

4.2.3 Predefined Groups

There are only two groups defined in the core SIE standard: group 0 blocks contain the SIE XML metadata and group 1 blocks are index blocks.

XML metadata (group 0)

The SIE XML metadata defines how to interpret the SIE data. It contains tags, channels and tests. Tags are generalized metadata relating an arbitrary text key to a completely arbitrary value. Channels are groupings of engineering data (composed of multiple dimensions) with their associated metadata. Tests are groupings of channels. Tags can appear at all levels of the hierarchy.
Additionally, the SIE XML metadata contains descriptions of all binary formats contained within the file, including the block structure defined above. These descriptions are contained within decoders, which are small programs used to read and interpret binary data. The XML representation of the SIE metadata is defined as the sequential concatenation of the payload of every block with group ID 0.

Index blocks (group 1)

Index blocks must index a consecutive chunk of blocks ending with the last block before the index block itself. Index blocks that are indexed in other index blocks will be passed over to allow re-indexing of a file.

4.2.4 Data Rendering Algorithm

The first step in getting the data model representation of an SIE channel's data is to look at the channel element (`ch`) defining the channel's metadata.
The following information must be obtained for each dimension d: which group the data are going to be read from (g), which decoder will be used to decode the data (D), which `v` of the decoder will be assigned to the dimension (v_d), and optionally, what transform will be applied to the output of the decoder (x_d). Note that for this SIE version, the group and decoder for all dimensions must be the same, as there is no

way to specify how multiple disparate decoder outputs be joined. If any of the required information above is missing, the channel is abstract and does not have data associated with it. This is common in the case of base channels which are only used to hold common metadata for other channels. Usually, these channels will have the private attribute set so the reader knows not to try to access them.
To determine the group, look at the group attribute for each of the dimension elements, or the group attribute of the channel element. The decoder ID is defined by the decoder attribute in each dimension's data element and the decoder v by the `v` attribute. Finally, the transform is defined by the dimension's `xform` element, if any. Knowing g, D, all v_d, and all x_d, the algorithm to render the channel data is relatively straightforward:

```
For each block $b$ in the SIE stream with group ID $g$:
  Run decoder $D$ on the payload of $b$, producing multiple output vectors.
  For each decoder output vector $V$:
    Start building up an output row $r$.
    For each dimension $d$ in the channel:
      Let $o$ be the decoder output $V[v_d]$.
      If the transform $x_d$ exists, apply it to $o$.
      Assign $r[d]$ to be the value of $o$.
    Append the row $r$ to the channel data.
```

The result of this algorithm will be the channel data in the form of the universal SIE data model.

4.3 XML Details

4.3.1 Standard Preamble

The standard SIE XML metadata preamble is:

```
<?xml version="1.0" encoding="UTF-8"?>
<sie version="1.0" xmlns="http://www.somat.com/SIE">
<!-- SIE format standard definitions: -->
  <!-- SIE stream decoder: -->
  <decoder id="0">
    <loop>
      <read var="size" bits="32" type="uint" endian="big"/>
      <read var="group" bits="32" type="uint" endian="big"/>
      <read var="syncword" bits="32" type="uint" endian="big"
            value="0x51EDA7A0"/>
      <read var="payload" octets="{$size - 20}" type="raw"/>
      <read var="checksum" bits="32" type="uint" endian="big"/>
      <read var="size2" bits="32" type="uint" endian="big"
            value="{$size}"/>
    </loop>
  </decoder>
<tag id="sie:xml_metadata" group="0" format="text/xml"/>

  <!-- SIE index block decoder:  v0=offset, v1=group -->
  <decoder id="1">
    <loop>
```

```
        <read var="v0" bits="64" type="uint" endian="big"/>
        <read var="v1" bits="32" type="uint" endian="big"/>
        <sample/>
      </loop>
    </decoder>
    <tag id="sie:block_index" group="1" decoder="1"/>

  <!-- Stream-specific definitions begin here: -->
```

Notice this contains a description of the block structure described in the previous chapter and the tag `sie:xml_metadata`, the value of which is the contents of group 0. It also contains a tag and description of the format of the block indexes.
The purpose of including this preamble is not to expect that a reader program will extract how to read the SIE block structure from this description; a sufficiently clever implementation could, but that value of this is questionable. Rather, it is to realize the design goal that the format be completely self-describing. Even without external documentation, looking at an SIE file in a text editor will yield a good chance of understanding the basic format.

4.3.2 XML Features

To make possible some of the properties of the SIE format, the XML representation used has some unusual features.

Merge and Replace Behavior

To support streaming data, the XML format supports effectively overriding and modifying metadata from earlier in the stream. For example, the following two examples are equivalent.
Example 1:

```
<ch id="42" name="test">
  <tag id="core:description">testing</tag>
</ch>
<ch id="42">
  <tag id="core:output_samples">74088</tag>
</ch>
```

Example 2:

```
<ch id="42" name="test">
  <tag id="core:description">testing</tag>
  <tag id="core:output_samples">74088</tag>
</ch>
```

This is an example of merging behavior. This can be used to add information once it is known. In the example above, the number of output samples is clearly not known until all samples are written to the file.
Some other metadata are atomic and replace instead as illustrated by the following two examples.
Example 1:

```
<ch id="42" name="test">
  <tag id="core:description">testing</tag>
</ch>
```

```
<ch id="42">
  <tag id="core:description">overridden</tag>
</ch>
```

Example 2:

```
<ch id="42" name="test">
  <tag id="core:description">overridden</tag>
</ch>
```

The table below details the merge and replace behavior for interpreting the SIE XML metadata. The behavior is either merge or replace, as described above. The key is the attribute which is used to select which existing element to merge with or replace. If the key is unique, only one element of that type is allowed within its parent.

Element	Behavior	Key
`ch`	merge	`id`
`dim`	merge	`index`
`test`	merge	`id`
`data`	replace	unique
`tag`	replace	`id`
`xform`	replace	unique

Nesting Shortcut

To reduce metadata size and require fewer statements from the writer, there is a shortcut method to describe elements nested within others. The attribute names `test`, `ch` and `dim` are reserved for all elements and produce hierarchy as illustrated by the following examples.

Example 1:

```
<tag ch="2" dim="3" test="1" id="core:description">test</tag>
```

Example 2:

```
<test id="1">
  <ch id="2">
    <dim index="3">
      <tag id="core:description">test</tag>
    </dim>
  </ch>
</test>
```

Note that to allow this to be implemented by a simple expansion, there cannot be redundant element specifiers. For example, the following would be in error, as it creates two nested `test` tags:

```
<test id="1">
  <tag ch="2" dim="3" test="1" id="core:description">test</tag>
</test>
```

Inheritance
The channel and test elements support inheritance. To inherit, set the `base` attribute to the ID of the existing element to inherit from. For instance, the following two examples are equivalent.
Example 1:

```
<ch id="2" name="old">
  <tag id="core:description">test</tag>
</ch>
<ch id="42" name="new">
  <tag id="core:description">test</tag>
</ch>
```

Example 2:

```
<ch id="2" name="old">
  <tag id="core:description">test</tag>
</ch>
<ch id="42" base="2" name="new"/>
```

Inheritance is useful for creating many similar channels without excessive duplication.

4.3.3 XML Metadata Grammar

NOTE
The syntax of each element is described in the RELAX NG schema description language.

SIE Element
The SIE element is the top-level element of the SIE XML metadata. The `version` attribute specifies the version of SIE. Version 1.0 is described in this document. The SIE element can contain tags, decoders, channels and tests.

```
SIE = element sie {
  attribute version { text },
  ( TopTag | Decoder | Channel | Test )*
}
```

Note that as an SIE stream is always capable of being appended to, the SIE element should never be explicitly closed. For parsing purposes, consider “`</sie>`” added to the end of the XML metadata.

Tag Element
The tag element is the generic element for application-level annotation, which allows arbitrary text as the tag name and completely arbitrary values. Tag values are stored either directly in the XML, properly escaped:

```
<tag id="core:samplerate">500</tag>
```

or as the contents of a group:

```
<tag id="setupphoto.1" group="27" format="image/jpeg"/>
```

For the group form, the value of the tag is considered to be the sequential concatenation of the payload of every block with the specified group ID.
The `<tag group="42" decoder="17"/>` form allows binary structures specifiable through the decoder mechanism. Extant SIE reader implementations are not capable of applying decoders to tag values.

```
TestSpecs = ( ChannelSpec | ( ChannelSpec & DimSpec ) )

TopTag = element tag {
  (attribute test { UINT } | TestSpecs)?,
  TagBase
}

TestTag = element tag {
  TestSpecs?,
  TagBase
}

ChannelTag = element tag {
  DimSpec?,
  TagBase
}

Tag = element tag {
  TagBase
}

TagBase = (
  attribute id { text },
  ((attribute group { UINT }, DecoderOrFormat?) | text)
)

DecoderOrFormat = ( attribute decoder { UINT } | Format )

UINT = xsd:nonNegativeInteger
```

Decoder Element
Each decoder element contains the machinery to decode the binary payload from a particular data block. The same decoder may be used by any number of different channels. The assignment of a particular decoder ID to a group is mediated through the group and channel directives.

```
Decoder = element decoder {
  Id,
  DecodeOps*
}

DecodeOps = ( If | Loop | Read | Sample | Seek | Set )
```

Variables
Decoder variables can contain either numbers or arbitrary binary data of unbounded size. Variables are always initialized to zero upon entering a decoder. The special variables `v0`, `v1`, ... , `v`*n* are used by the sample operator to compose output result vectors.

Expressions

Expressions support simple arithmetic expressions and variable dereferencing via the `$` prefix (i.e. the value of `{$foo + 24}` is 24 plus the value of the variable *foo*).

```
EXPR = text # either a literal number or an {...} expression
```

If Operator

The if operator is used for conditional execution of code. If the expression in the `condition` attribute evaluates to non-zero, the contents of the if operator are executed.

```
If = element if{
  attribute condition { EXPR },
  DecodeOps*
}
```

Loop Operator

The loop operator is a general iteration mechanism. By default, it repeats its contents forever or until the decoder aborts upon a failed read operation.
If the variable attribute is present, the optional `start`, `end` and `increment` attributes can be specified as either a constant value or an expression. The variable attribute (`var`) begins the loop equivalent to the value of the `start` attribute as illustrated by the following two equivalent statements.

```
<set var="foo" value="42"/><loop var="foo">
<loop var="foo" start="42">
```

The loop variable increments by the `increment` value at the end of each loop iteration. The `increment` value defaults to one. The `end` attribute sets the termination condition; the loop runs while the `var` is less than `end` when the `increment` is zero or positive, and while `var` is more than `end` when the `increment` is negative. If defined as an expression, the `increment` and `end` values are computed with each iteration.

```
Loop = element loop {
  (attribute var { text }, LoopOpts)?,
  DecodeOps*
}
LoopOpts = (
  attribute increment { EXPR }?,
  attribute start { EXPR }?,
  attribute end { EXPR }?
)
```

Read Operator

The read operator reads *n* bits of payload from the current position, where *n* is either the value of the `bits` attribute or eight times the value of the `octets` attribute. If used, `bits` must be a multiple of eight. If neither size attribute is present, all remaining data in the payload is read. If there is less data remaining than requested, the decoder terminates; this is the primary means of termination for most decoders.

If the variable attribute (`var`) is present, the named variable is set to the value `read` and interpreted according to any `type` or `endian` attributes that may be present. The optional `value` attribute asserts that the value read is equal to the value of the attribute. An error will result if this is not the case.
The `type` attribute, which defaults to raw, has several possible values:

Type	Description
int	Two’s complement signed integer of 8, 16, 32 or 64 bits.
uint	Unsigned integer of 8, 16, 32 or 64 bits.
float	IEEE-754 floating point value of either 32 or 64 bits.
raw	No interpretation; store unmodified binary buffer. The `endian` attribute has no effect here.

The `endian` attribute specifies the byte order to be used for interpreting numerical data. If the endian attribute is “big,” the number `0x00112233` would be stored in the order `0x00`, `0x11`, `0x22`, `0x33` (also known as network or Motorola order). If the endian attribute is “little,” it would be stored in the order `0x33`, `0x22`, `0x11`, `0x00` (Intel, Vax, x86 order).

```
Read = element read {
  attribute var { text }?,
  ( attribute bits { EXPR } | attribute octets { EXPR }),
  ReadType,
  attribute value { EXPR }?
}
ReadType = (
  Format
  | attribute type { "raw" }
  | ( attribute type { "int" | "uint" | "float" }
  & attribute endian { "big" | "little" })
)
```

Sample Operator
The sample operator adds a data sample vector [`v0`,`v1`, ... ,`v`*n*] to the decoder’s output queue. All variables of the form v0, v1, v2, etc. that are used in the decoder (through loop, set or read operators) are included in the output vector.

```
Sample = element sample { empty }
```

Seek Operator
The seek operator moves the location from which the next read operator will read data. The read location is moved to the position described in the `from` and `offset` attributes. The `from` attribute can either be “start,” the start of the data; “current,” the current position; or “end,” the end of the data. The `offset` attribute is an expression evaluating to the number of octets offset from the location defined in the `from` attribute.

```
Seek = element seek{
  attribute from { "start" | "current" | "end" },
  attribute offset { EXPR }
}
```

Set Operator

The set operator is used to set the variable specified by the variable attribute (`var`) to the value of the `value` attribute. The `value` attribute can be either a numerical value or an expression.

```
Set = element set{
  attribute var { text },
  attribute value { EXPR }
}
```

Channel (`ch`) Element

The channel element (`ch`) is the container used to define engineering level data channels. It contains dimension elements whose transform, data and tag elements have all the information necessary to convert the raw decoder data vectors into engineering unit data. The optional `name` attribute defines the channel name. All other information is contained in tag elements either directly under the channel or under a dimension child.

To minimize redundancy, the channel element supports an inheritance mechanism through its `base` attribute. See "Inheritance" on page 37 for more information on the `base` attribute.

The optional `group` attribute specifies a single group to be associated with the channel. This will be most often seen using inheritance inside a test definition:

```
<test id="1">
  <ch id="9" base="0" group="5" name="th1@rpm.RN2"/>
  <ch id="10" base="1" group="5" name="th1@coolant.RN2"/>
</test>
```

More than one group can contribute to a channel. The `group` attribute in a dimension overrides any `group` attribute of the parent channel.

```
Channel = element ch{
  Id,
  attribute base { UINT }?,
  attribute name { text }?,
  attribute group { UINT }?,
  Private?,
  (Dimension* & ChannelTransform* & ChannelTag*)
}
```

Dimension (`dim`) Element

The dimension element (`dim`) contains the definition for axis `index`. The first axis is index zero. Decoder data for the dimension comes from the dimension's `group` attribute or, if that does not exist, the enclosing channel's `group` attribute.

```
Dimension = element dim {
  attribute index { UINT },
  attribute group { UINT }?,
  ( Transform? & Data? & Tag* )
}
```

Transform (`xform`) Element

The transform element (`xform`) defines a linear transform using the `scale` (i.e. slope) and `offset` (i.e. intercept) attributes. An indexed mapping transform (essentially an array lookup) uses `index_ch` and `index_dim` to define the output.

```
Transform = element xform { Transforms }

TestTransform = element xform { ChannelDimSpec, Transforms }

ChannelTransform = element xform { DimSpec, Transforms }

Transforms = (LinearTransform | IndexTransform)

LinearTransform = (
  attribute scale { REAL },
  attribute offset { REAL }
)

IndexTransform = (
  attribute index_ch { UINT },
  attribute index_dim { UINT }
)
```

Data Element

The data element is used to define the data for a particular dimension as the vector element *v* from decoder *d*. For example,

```
<dim index="2">
  <data decoder="7" v="3">
</dim>
```

assigns element “3” of the decoder “7” output to dimension “2” of the channel.

```
Data = element data{
  DecoderOrFormat,
  attribute v { UINT }
}
```

4.4 Core Schema

SIE provides a very flexible framework for describing almost any kind of data and associated metadata. However, some organization and standardization is necessary to allow the widest number of programs to understand the widest amount of data.

SIE tag names can belong to a particular metadata schema. The schema part of a tag name precedes a colon. For example, tag names beginning with `core:` belong to the core schema.

Not all of the tags in the schema will be present; in fact, none have to be. For maximum robustness, data reading applications should be able to work with the absolutely minimum set of tags possible.

4.4.1 Core Metadata Schema

In addition to metadata schemas, the core schema tag (see “Schema Tag” on page 44) indicates which data schema is in use for the numerical and binary data that will be output from a channel. Referencing the correct data schema documentation will allow the data to be correctly interpreted.

`core:description`

Description Tag
The core description tag provides a free-form, natural-language description of the element in which it is contained.

```
<tag id="core:description">
  A thermocouple mounted underneath the right front wheel
bearing.
  This channel is gated to only collect data when the
temperature   is above 60 degrees C.
</tag>
```

`core:elapsed_time`

Elapsed Time Tag
The core elapsed time tag defines the amount of time spent collecting the data in its container in seconds.

```
<tag id="core:elapsed_time">160760.4</tag>
```

`core:input_samples`

Input Samples Tag
The core input samples tag contains the number of data samples present before running any data-reduction algorithms. For example, in a gated time history, the input samples tag contains the number of samples present before gating.

```
<tag id="core:input_samples">4760</tag>
```

`core:label`

Label Tag
The core label tag defines a label, which is a short description intended for placing on a plot, graph or table. For example, a dimension label is usually used as an axis label.

```
<tag id="core:label">Time</tag>
```

`core:output_samples`

Output Samples Tag
The core output samples tag contains the number of data samples that are present after running any data-reduction algorithms or, in other words, the number of data samples that are actually stored in the channel. For example, in a gated time history, the output samples tag contains the number of samples stored after gating.

```
<tag id="core:output_samples">200</tag>
```

`core:range_max`

Range Max Tag
The core range max tag defines the expected maximum bound for the container dimension. This can be used, for example, to set plot range boundaries. It is not the maximum data value in the dimension.

```
<tag id="core:range_max">1000.0</tag>
```

`core:range_min`

Range Min Tag
The core range min tag defines the expected minimum bound for the container dimension. This can be used, for example, to set plot range boundaries. It is not the minimum data value in the dimension.

```
<tag id="core:range_min">-1000.0</tag>
```

`core_sample_rate`

Sample Rate Tag
The core sample rate tag defines the sample rate in hertz of the data in its container.

```
<tag id="core:sample_rate">2500</tag>
```

`core:schema` **Schema Tag**

The core schema tag defines what data schema is in use for this channel. The data schema defines how you interpret the data output of the channel. For example, if the schema tag is `somat:sequential`, look up the sequential data schema for instructions on how to interpret it. For more information on the SoMat data schemas, see "SoMat Data Schema" on page 46.

```
<tag id="core:schema">somat:sequential</tag>
```

`core:setup_name` **Setup Name Tag**

The core setup name tag contains the name of the setup under which the data in the container is collected, if applicable.

```
<tag id="core:setup_name">bearing_temp</tag>
```

`core:start_time` **Start Time Tag**

The core start time tag contains the time that data collection starts for the container. The time is in the ISO 8601 format.

```
<tag id="core:start_time">2007-01-10T11:49:34-0600</tag>
```

`core:stop_time` **Stop Time Tag**

The core stop time tag contains the time that data collection stops. The time is in the ISO 8601 format.

```
<tag id="core:stop_time">2007-01-10T12:12:06-0600</tag>
```

`core:test_count` **Test Count Tag**

The core test count tag contains the sequence number of the container test. For example, when a test is repeated multiple times, the first test run has a test count of 1, the second has a test count of 2 and so on.

```
<tag id="core:test_count">14</tag>
```

`core:units` **Units Tag**

The core units tag defines the units of the container dimension (e.g. seconds, millivolts, microstrain, etc.). For unitless dimensions (e.g. counts) this tag is absent.

```
<tag id="core:units">seconds</tag>
```

`core:version` **Version Tag**

The core version tag defines the version of the core schema in use. The version described in this document is 1.0.

```
<tag id="core:version">1.0</tag>
```

4.5 SoMat Schema

This section describes the SoMat schema for the SIE format. The SoMat schema describes metadata and data representations specific to HBM's line of SoMat data acquisition systems. In addition, some elements of this schema may be applicable for more general use. In almost all cases, data files that reference this schema also reference the core schema.

4.5.1 SoMat Metadata Schema

somat:data_bits

Data Bits Tag
The SoMat data bits tag contains the number of bists of resolution of the channel data..

```
<tag id="somat:data_bits">16</tag>
```

somat:data_format

Data Format Tag
The SoMat data format tag contains the data format of the channel data type. The data format can equal “uint,” “int” or “float.”

```
<tag id="somat:data_format">uint</tag>
```

somat:datamode_name

DataMode™ Name Tag
The SoMat DataMode name tag contains the name of the DataMode which produced the current SIE channel.

```
<tag id="somat:datamode_name">th1k</tag>
```

somat:datamode_type

DataMode™ Type Tag
The SoMat DataMode type tag contains the type of the DataMode produced the current SIE channel.

```
<tag id="somat:datamode_type">time_history</tag>
```

The data mode type can be one of the following values:

```
time_history
burst_history
peak_valley
peak_valley_slice
event_slice
time_at_level
peak_valley_matrix/from_to
peak_valley_matrix/range_mean
peak_valley_matrix/range_only
rainflow/from_to
rainflow/range_mean
rainflow/range_only
message_log
```

somat:input_channel

Input Channel Tag
The SoMat input channel tag contains the name of the input channel which produced the current SIE channel.

```
<tag id="somat:input_channel">bracket</tag>
```

somat:log

Log Tag
The SoMat log tag contains the data acquisition system’s log. The log is stored continuously and is not limited in size.

somat:rainflow_unclosed_cycles

Rainflow Unclosed Cycles Tag
The SoMat rainflow unclosed cycles tag contains a sequence of ASCII-formatted floating-point numbers containing the unclosed cycles stack from the rainflow counting algorithm.

```
<tag id="somat:rainflow_unclosed_cycles">
  9.992119789123535 20.02879905700684 -40.09270095825195
  40.0364990234375 -50.09659957885742 60.07699966430664
-  70.13710021972656 70.08080291748047 -80.14089965820312
  90.12129974365234 -110.1849975585938 150.2100067138672
-  120.1890029907227 190.2579956054688 -150.2330017089844
  230.3390045166016 -330.5 390.5650024414062
-681.0609741210938   630.9199829101562 -590.89501953125
460.6900024414062 -  460.7139892578125 340.5130004882812
-320.4960021972656   170.2510070800781 -210.3220062255859
</tag>
```

somat:tce_setup

TCE Setup Tag
The SoMat TCE setup tag contains the TCE setup file used to initialize the current test.

somat:version

Version Tag
The SoMat version tag defines the version of the SoMat schema in use. The version described in this document is 1.0.

```
<tag id="somat:version">1.0</tag>
```

4.5.2 SoMat Data Schema

somat:sequential

Sequential Data Schema
The SoMat sequential data schema represents time series numerical data sampled at a regular interval or the sequential output of various data reduction algorithms which dispose of time information. Each row represents a single data sample and the time or sequence number of that sample.
When time is preserved, the data output represents:

SIE dimension	Data type	Scaled	Unscaled
0	Numeric	Time	Sample number
1	Numeric	Engineering value	undefined

When time information is disposed of, the data output represents:

SIE dimension	Data type	Scaled	Unscaled
0	Numeric	Sequence number	Sequence number
1	Numeric	Engineering value	undefined

The values of SIE dimension index 0, scaled or unscaled, have non-decreasing ordering.

somat:message

Message Data Schema
The SoMat message data schema data schema represents non-numerical data sampled at irregular intervals. Each row represents a single message or event and the time of collection.

SIE dimension	Data type	Scaled	Unscaled
0	Numeric	Time	undefined
1	Raw	Binary message	undefined

The values of SIE dimension index 0, scaled or unscaled, have non-decreasing ordering.

somat:burst

Burst Data Schema
The SoMat burst data schema represents time series numerical data sampled at a regular interval, but where actual data collection is triggered by a triggering event. Each row represents a single data sample, the time that sample was collected, and the relation between that sample and the event which triggered collection.

SIE dimension	Data type	Scaled	Unscaled
0	Numeric	Time	Sample number
1	Numeric	Engineering value	undefined
2	Numeric	Burst index	Burst index

The values of SIE dimension index 0, scaled or unscaled, have non-decreasing ordering.
For the burst data schema, the sample number is absolute. For example, if the first burst data happens at the 50,000th sample collected, the sample number emitted from SIE dimension 0 for that data is 50,000, not 0. The burst index, however, indicates the relative position of the burst trigger.

Burst index	Description
$n<0$	Current sample is n samples before the burst trigger.
0	Current sample is the first sample in the burst trigger.
0.5	Current sample is a continuation of the (level-sensitive) burst trigger.
$n\geq1$	Current sample is n samples after the burst trigger.

somat:histogram

Histogram Data Schema
The SoMat histogram data schema represents n-dimensional histogram data. Each row represents a single histogram bin's count and limits in all incoming dimensions. The histogram bins are not presented in any particular order, nor are empty bins guaranteed to be present. The number of histogram dimensions is equal to $(m-1)/2$, where m is the number of SIE dimensions. To allow streaming of histogram data, if a bin is specified more than once, the last count is valid.

SIE dimension	Data type	Scaled	Unscaled
0	Numeric	Bin count	undefined
n*2+1	Numeric	Lower bin limit for histogram dimension n	undefined
n*2+2	Numeric	Upper bin limit for histogram dimension n	undefined

A data sample falls into a bin if it is in the range [*lower*, *upper*). Note that the lower bound is closed while the upper bound is open. Overflow bins are explicitly specified; a negative overflow bin has a lower bin limit of negative infinity, while a positive overflow bin has an upper bin limit of infinity.
The libsie library offers a convenient interface to access a histogram stored in this schema as an *n*-dimensional array rather than as a linear list of bins. For more information, see “Histogram Access” on page 28.

`somat:rainflow`

Rainflow Data Schema

The SoMat rainflow data schema is identical to the histogram data schema except for the SoMat rainflow unclosed cycles metadata tag (`somat:rainflow_unclode_cycles`) metadata tag. This tag contains a sequence of ASCII-formatted floating-point numbers containing the unclosed cycles stack from the rainflow counting algorithm. Note that histograms emitted by SIE are closed. To reopen them, run the rainflow algorithm in reverse with the unclosed cycles stack.

Europe, Middle East and Africa
HBM GmbH
Im Tiefen See 45
64293 Darmstadt, Germany
Tel: +49 6151 8030 • Email: info@hbm.com

The Americas
HBM, Inc.
19 Bartlett Street
Marlborough, MA 01752, USA
Tel: +1 800-578-4260 • Email: info@usa.hbm.com

Asia-Pacific
HBM China
106 Heng Shan Road
Suzhou 215009
Jiangsu, China
Tel: +86 512 682 47776 • Email: hbmchina@hbm.com.cn

I2954-1.0 en SoMat P/N DOC 0024-00

measure and predict with confidence

HBM