LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY
- LIGO -
CALIFORNIA INSTITUTE OF TECHNOLOGY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Document Type	**LIGO-T990030-v2**	2010/03/25

LIGO Scientific Collaboration Algorithm Library Specification and Style Guide

Bruce Allen, Kent Blackburn, Duncan Brown, Jolien Creighton, Teviet Creighton, Sam Finn, Albert Lazzarini, Adam Mercer, and Alan Wiseman

Distribution of this draft:

LIGO Scientific Collaboration
DRAFT

California Institute of Technology
LIGO Project - MS 51-33
Pasadena CA 91125
Phone (626) 395-2129
Fax (626) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project - MS NW22-295
Cambridge, MA 01239
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

WWW: http://www.ligo.caltech.edu/

RCS Id — Processed with LaTeX on 2010/03/25

Contents

1 Introduction

1.1 The LSC Algorithm Library

The LSC Algorithm Library, hereafter LAL, is a library of routines for use in gravitational wave data analysis.

1.2 The goal of the LAL software specification

From the first edition of this guide: **The defining purpose of this document is to establish a software specification that fosters widespread-use and collaborative-development of a well-tested analysis library.** The goal is to develop a *portable* and *convenient* library, both for users and developers.

To achieve portability, the library is a library of routines written in a subset of C99, which is well supported on nearly every modern computing environment, and the routines can easily be used by programs written in other languages (C++, Fortran, Python, etc.).

The first edition of this specification contained several highly restrictive rules that were conceived to give LAL routines a standard "look and feel" to promote their ease of use and also to promote good programming practice. Unfortunately, some of these rules turned out to be counterproductive, making some routines have an awkward interface and encouraging large, monolithic functions, i.e., promoting poor programming practices.

This second edition makes several significant changes to the original specification in an attempt to alleviate the most egregious over-restrictive rules. However, since there is already a large existing code-base, it is not possible to completely re-write the specification. The new specification must be consistent with the old conventions. Therefore this new specification still retains many of the elements of the old specification.

And sometimes for good reason! Many of the rules in the original specification, while perhaps being somewhat restrictive and burdensome on the developer, were nevertheless very important for the concept of a portable data analysis algorithm library (and did represent good programming practices).

The more rigid rules in this document are set aside in boxes with red type. Here is the first rigid rule:

> The rules in this specification may need to be periodically reexamined. Therefore, this is a living document. The librarian will amend this document as needed. Significant changes to the document will be made in consultation with the LSC Software Change Control Board.

1.3 The elements of the library specification

The library specification has the following elements:

1. A coding style, which is needed in order to establish a library namespace (so that routines in the LAL library can be used in conjunction with other libraries), and to maintain a (somewhat) uniform look-and-feel.

2. Function requirements, which are needed in order to maintain portability and to establish a (relatively) standard API.

3. Standard data structures, macros, and functions, which assist in providing a common set of tools for developers to promote uniformity (where desired) and collaborative development.

1.4 The LAL and XLAL interfaces

Because the original LAL specification was relatively "heavy-weight," (in that there was significant overhead required for new functions), LAL functions tended to be large and monolithic, and often a particular "routine" was re-written many times in-line in many different functions. It is not possible to simultaneously maintain the semblance of the original LAL requirements on functions and completely remedy this deficiency. Instead, this specification introduces a second, parallel interface, called the XLAL interface, specifically for writing small, light-weight, helper routines. The intent is for the LAL interface to be the primary interface for users of the LAL library, but for the XLAL interface to be a convenient interface for use within the LAL library. (Think of the "X" being an underscore.)

There are some rules that apply to both the LAL and the XLAL interfaces, and these will be described first. Then the rules that are specific to the two interfaces will be described in separate sections.

All functions in the LAL library that have external linkage shall be either LAL functions or XLAL functions. The LAL functions shall conform to the general rules for functions and to the rules specific to LAL functions. The XLAL functions shall conform to the general rules for functions and to the rules specific to XLAL functions.

2 Coding style guidelines

2.1 Atomic data types

For historical reasons more than anything I can think of, LAL routines should use the LAL-specific atomic data types (`REAL4` rather than `float`, `REAL8` rather than `double`, `CHAR` rather than `char`, `UCHAR` rather than `unsigned char`) and should use `INT2`, `UINT2`, `INT4`, `UINT4`, `INT8`, and `UINT8`, which have a platform-independent size, rather than `short int`, `unsigned short int`, `int`, `unsigned int`, `long int`, and `unsigned long int` (and especially not `long long` or `long double`) which do not. LAL makes certain requirements on these types. For example, `REAL4` and `REAL8` must be single and double precision IEEE-754 floating-point variables. Invariably they are equivalent to `float` or `double` on a given system (or else LAL won't work at all on that system). Similarly, a `INT4` is a four-byte integer (and it is assumed that each byte is eight bits on any system that LAL is installed), so it will be valid over the expected range.

Sometimes when the size of an integer variable is not crucial (e.g., for return codes from `XLAL` functions), `int` is used. It is also necessary to allow the `int` type in LAL for functions such as `frexp`. Standard C functions that have arguments that are pointers to type `double`, e.g., `modf`, can receive a pointer to type `REAL8` instead. That is, the type `REAL8` can be assumed to be equivalent to the type `double`.

2.2 Names of functions, variables, etc.

These rules are to define a standard namespace scheme. They apply to all functions with external linkage (i.e., those functions not preceded by the `static` keyword), as well as types, macros, etc., in header files.

1. All function names use StudlyCaps and begin with either `LAL` or `XLAL`, e.g., `LALExampleFunction`, `LALDoICare`, etc. Underscores are not used.

2. All types also use StudlyCaps and begin with a capital letter, e.g., `LALMyType`. Custom data structures must be given names that try to avoid namespace conflicts; we suggest simply prefixing the name with `LAL` or `XLAL` or with the name of one of the LAL atomic data types, e.g., `REAL4`.

3. Global variables **of which there are NONE** (except those specifically allowed by the Librarian), and fields within a structure or a union, are in studlyCaps beginning with a lower case letter. Global variables will begin with either `lal` or `xlal`, e.g., `lalDebugLevel`.

4. Macros are generally all in UPPERCASE and compound macro names may use underscores. As with the types, to avoid namespace collisions, it is recommended that the macro begin with `LAL_` or `XLAL_`.

5. Local variables can have any name that does not shadow a standard global symbol name (whether in LAL or in a standard C library or other likely names). Thus, do not call a variable `exit` or `LALMalloc` or even `pow`. And don't declare the variable `i` at the top level of a function and then shadow it in a block within that function. This is just good programming practice.

New data types will be declared as shown in this example for the data type `LALMyType`:

```
typedef struct
tagLALMyType
{
  INT4  firstField;
  REAL4 secondField;
}
LALMyType;
```

Note that the structure name is `tagLALMyType`.

2.3 Header and source file conventions

The LAL API is defined by the *installed* header files (there may be additional header files that are used when compiling LAL that are not installed, but these then do not form part of the API as they are not made available to the user). All functions and variables with external linkage as well as any datatypes, enumeration constants, macros, etc., that form part of the API must be defined in these installed header files. These installed header files will be installed in the location where header files normally reside on a system in a subdirectory called `lal`. All header files should include other LAL header files as follows: suppose that `LALThisHeader.h` needs to include `LALAnotherHeader.h`, then it should do so as follows:

```
#include <lal/LALAnotherHeader.h>
```

All header files should be idempotent. This means they need to have include guards. To do so, the first two lines of `LALThisHeader.h` should be something like:

```
#ifndef LALTHISHEADER_H
#define LALTHISHEADER_H
```

and the last line of the file should be

```
#endif /* LALTHISHEADER_H */
```

To be compatible with C++, all declarations should be wrapped as follows:

```
#ifdef __cplusplus
extern "C" {
#endif
```

⟨*declarations*⟩

```
#ifdef __cplusplus
}
#endif
```

It is important that all source files (header files, whether installed or not, and the C source files) contain the RCS ID information, which is then put into the LAL library so that it can be examined later. The convention for this is to have all `.h` have lines similar to these (for `LALThisHeader.h`):

```
#include <lal/LALRCSID.h> /* if no other LAL header has been included */
NRCSID( LALTHISHEADERH, "$Id$" );
```

near the top. Since all LAL header files will ultimately include `lal/LALRCSID.h`, it only explicitly needs to be included if no other LAL header has yet been included. Note that the string `"$Id$"` will be expanded by CVS into some string describing the current version of the file. Similarly, a `.c` file such as `LALThisSourceFile.c` would have the following

```
#include <lal/LALRCSID.h> /* if no other LAL header has been included */
NRCSID( LALTHISSOURCEFILEC, "$Id$" );
```

2.4 Language requirements

LAL code should all be in "clean C," i.e., that language that is a subset of both C and C++. This is not quite the same as just the C programming language. Only C-style comments should be used and avoid any constructs that would behave differently with C++-style comments. Names of variables, functions, etc., should not be any of the reserved keywords or names for the library. Some of the keywords and reserved names are listed here. The LAL namespace will assist in making sure that no conflicts arise. But even local variables names must be chosen carefully (e.g., so that they aren't the same as a C++ keyword). A list of keywords and reserved names, along with those standard C library functions that can be used, is found in appendix A

2.5 Filename conventions

Purely for the sanity of the Librarian, LAL has a rigid directory structure. LAL is composed of directories called packages whose names consist of entirely lower-case letters with no underscores. Within each package are four sub-directories called `doc`, `include` (which contains all the header files that are installed), `src` (which contains all the source files other than the installed header files but including those header files that are not installed), and `test`. Source files within these directories will be named with StudlyCaps (starting with a capital letter, no underscores). Refer to the LAL Software Documentation [1].

For documentation purposes, a *package* contains a set of *headers* (the installed headers), which contain prototypes for functions that are organized in *modules* of one or more functions, each module being a single `.c` file. Thus, all the functions with external linkage in a `.c` file must be prototyped in the same header file in the same package.

3 Common rules for both the LAL and the XLAL functions

3.1 Function arguments

Function arguments must be one of the following (atomic) types: `CHAR`, `UCHAR`, `INT2`, `UINT2`, `INT4`, `UINT4`, `INT8`, `UINT8`, `int`, `REAL4`, `REAL8`. In addition, functions may take a pointer as an argument. Structures or unions (including `COMPLEX8` and `COMPLEX16`) must not be passed directly to a function as an argument; pass a pointer instead. Arguments may be qualified with `const` if desired.

> All arguments to functions must be of one of the following types: `CHAR`, `UCHAR`, `INT2`, `UINT2`, `INT4`, `UINT4`, `INT8`, `UINT8`, `int`, `REAL4`, `REAL8`, or a pointer to any object. XLAL functions may also have no arguments (`void`), or a variable number of arguments of the above types (`...`).

3.2 Functions should not have any dependence on system environment

The first part of this rule is that functions should not do any file I/O since there should be no assumptions about the nature of the filesystem. LAL is not supposed to assume POSIX. Furthermore, there should be no assumptions about (or dependence on) the environment under which a LAL function is called. This will allow LAL routines to be integrated into a wide variety of programming environments: they may be used in stand-alone programs or in loadable modules integrated into other run environments. Specifically this means that routines in `stdio.h` are not allowed (except for `snprintf`), several routines in `stdlib.h` including `rand` and `srand` (there are LAL replacements for these), `system`, and `getenv`.

> Functions will not perform any file I/O or have any dependence on the system environment. Specifically the latter means that functions such as `system`, `getenv`, `rand`, `srand` will not be used.

3.3 Memory management

Memory should always be allocated or freed with one of the LAL custom memory managers: `LALMalloc`, `LALCalloc`, `LALRealloc`, and `LALFree`, and *not* with `malloc`, `calloc`, `realloc`, and `free`. The LAL memory managers have additional memory leak checking ability that will assist in debugging if the debug level is set appropriately.

> All memory allocation shall be done with the functions `LALMalloc`, `LALCalloc`, or `LALRealloc`, and shall be freed with the functions `LALFree` or `LALRealloc`; the functions `malloc`, `calloc`, `realloc`, and `free` shall not be used.

Also, routines should free all memory allocated in that routine *except* for the memory that is explicitly created by that routine, *even if the routine exits with a failure code*. This will prevent memory leaks. LAL provides a routine `LALCheckMemoryLeaks` (which should not be called from any LAL function—instead it is for users of LAL to put at the end of `main`) which will make sure that all memory allocated by `LALMalloc` (etc.) has been freed with `LALFree`.

In fact, it is a good idea not to allocate any temporary memory within a routine. All temporary memory needed for a routine should be allocated by LAL functions that are designed for that purpose. Hence there are usually three classes of LAL functions:

- `LALCreateFoo` or `LALInitFoo` functions which create and initialize storage `foo` that the use will pass to...
- `LALBar` functions which uses the storage `foo`, and then the user calls...
- `LALDestroyFoo` or `LALFinalizeFoo` functions which destroys the storage in `foo`.

3.4 Functions must be reentrant and thread-safe

This rule essentially requires a functions behavior to depend only on its arguments. There should be no state saved in static storage within the function. That is, never use the `static` keyword within a function. In addition, all variables used by a function must be local to the function. That is, no global variables are allowed. (There are a few exceptions to this, e.g., the *reading* of the global `lalDebugLevel` variable, but these types of exceptions are under the control of the LAL librarian.)

Furthermore, use of routines that would cause a function to fail to be reentrant and thread-safe are not allowed. For example, many of the `time.h` routines (`asctime`, `ctime`, `gmtime`, `localtime`), some of the `string.h` routines (`strerror` and `strtok`), and other routines that are prohibited elsewhere for additional reasons.

> All functions will be reentrant and thread-safe. All local variables must be automatic. No global variables will be used. Routines such as `asctime`, `ctime`, `gmtime`, `localtime`, `strerror`, and `strtok` shall not be used as they are not reentrant and threadsafe.

3.5 Functions should always return control to the calling program

That is, routines should never explicitly raise signals, abort, or call the `exit` function, nor should they call functions that might do so. Also, since LAL is a library, don't change the behavior of `exit` or signals. Thus the routines `exit`, `atexit`, and routines in `signal.h` and `assert.h` should not be used. (But note the exception that XLAL functions do call an error handler, which can be set outside of the library to abort or exit.) Also, long-jumps are not allowed, so any routine in `setjmp.h` is not allowed.

> All functions will return control to the calling function. Functions such as `exit`, `atexit`, `raise`, `assert`, `abort` shall not be used. Long-jumps shall not be used.

4 Rules for LAL functions

All LAL functions must return `void` and have as their first argument a pointer to a `LALStatus` structure type. Any number of arguments may follow the status structure, though it is good style to be economical and to group miscellaneous data into structures where useful. The general convention is to have the first argument following the status structure to be the primary output from the function (i.e., a pointer to the result that is not used as input to the function). This general convention is for the convenience of the user who will come to appreciate that arguments are typically ordered in LAL as

```
void LALREAL4Divide( LALStatus *status, REAL4 *result, REAL4 numerator,
    REAL4 denominator )
```

rather than the following

```
void LALREAL4Divide( LALStatus *status, REAL4 numerator, REAL4 denominator,
    REAL4 *result )
REAL4)
```

> All LAL functions shall have names that begin with `LAL` followed by an uppercase letter.

> All LAL functions shall have no return value (type `void` return).

All LAL functions shall have a pointer to a LALStatus structure as their first argument. The contents of the LALStatus structure will be populated appropriately to indicate success or failure of the function call. The LALStatus structure is a linked list. If a LAL function (the *sub-function*) that is called from within a LAL function fails (the *top-function*), the status structure returned by the sub-function shall be the next element in the linked list of status structures returned by the top-function.

The status structure is maintained from the calling program and keeps a trace of all levels of LAL functions being called (it is a linked list of status structures). If a failure occurs, the status structure can be used to identify where and which sequence of functions have been called. The status structure is central to the "LAL interface." The status structure is not typically manipulated by hand... LAL provides several status handling macros for manipulating the status structure and reporting errors. The use of these macros imposes additional conventions on writing LAL functions. See the LAL Software Documentation [1] for a complete description of these conventions. As a brief synopsis, this is what a the source code for a simple LAL function such as LALREAL4Divide (in file LALDivide.c) might be:

```
#include <lal/LALDivide.h>
NRCSID( LALDIVIDEC, "$Id$" );

void
LALREAL4Divide(
    LALStatus *status,
    REAL4     *result,
    REAL4      numerator,
    REAL4      denominator
    )
{
   INITSTATUS( status, "LALREAL4Divide", LALDIVIDEC );
   ASSERT( result != NULL, status, LALDIVIDEH_ENULL, LALDIVIDEH_MSGENULL );
   if ( denominator == 0.0 )
     ABORT( status, LALDIVIDEH_EDIV0, LALDIVIDEH_MSGEDIV0 );
   *result = numerator / denominator;
   RETURN( status );
}
```

Here the error codes LALDIVIDEH_ENULL and LALDIVIDEH_EDIV0 and the corresponding error messages LALDIVIDEH_MSGENULL and LALDIVIDEH_MSGEDIV0 would be defined in LALDivide.h. The INITSTATUS structure is the first line of a LAL function. It populates the status structure with useful information such as the function name and RCS ID (which are the macro arguments). The RETURN macro prepares the status structure to indicate a nominal completion of the function; it should be used with any successful return. Error handling in this example is accomplished using either the ASSERT or ABORT macros. The ASSERT macros are usually used to check the sanity of arguments; the first macro argument is the result of a test that should be true otherwise the ASSERT macro will populate the status structure with an error code and message (specified by the third and fourth macro arguments) and will return from the function. The ASSERT functions are useful during debugging and development of code, and they are removed when LAL is compiled in production mode, so they can be used liberally. Thus true failures are captured instead with the ABORT macro. Like ASSERT, ABORT populates the status structure with an error code and message and returns, but it does not get removed when LAL is compiled in production mode. The ABORT macro is the normal way of dealing with error conditions.

There are several other status structure macros that are needed when preparing a status structure within a LAL function for calling another LAL function, for checking the result of that function call, and for handling situations when memory needs to be cleaned up before the function exits. The conventions for these situations are all described in the LAL Software Documentation. Here we will just note that these macros should always be used, and that a LAL function should never declare its own LALStatus structure for use when calling other LAL functions... the status structure used must always be one atta(t)ched to the provided status structure (so that the function call trace is maintained).

5 Rules for XLAL functions

The goal is to have XLAL functions be as flexible as possible in their interface while still requiring strict rules on error reporting. The XLAL functions are intended to be "lightweight" functions that can be used internally within the LAL

library. They don't have some of the burdens of LAL functions. In particular, they do not have a status structure. This immediately implies to the following:

1. **XLAL functions cannot call LAL functions.** Since LAL functions require a status structure, and since this status structure must initiate in the top-level program that interfaces with the LAL library (so that a trace of function calls is returned), XLAL functions cannot call LAL functions, even by having a local status structure. If you need to call a LAL function internally, the function must be a LAL function.

2. **XLAL functions can be "lightweight."** Initializing the status structure, atta(t)ching new structures to the list, setting the various fields of the status structure to indicate successes or failures, etc., can be somewhat burdensome for both the programmer and for the computer. The lack of a status structure in XLAL functions will relieve some of this burden and will hopefully allow for some of the tasks that are now done in large, monolithic code blocks to be divided into smaller and more modular XLAL functions. Compilers can then easily optimize code either by inlining the XLAL function or not depending on issues that the compiler understands (e.g., costs of a function call vs. cache misses, etc.).

3. **XLAL functions must report success or failure in other ways.** This puts some more burden on the developers to (i) make sure that the XLAL function correctly reports errors and (ii) understand how particular XLAL functions report their errors and deal with these appropriately. The goal is to design some rules for the XLAL functions that try to approach some degree of uniformity without overly hampering their interface.

For the purpose of providing a relatively uniform error reporting system, it is necessary to categorize XLAL functions into four likely types, which are based on their return types.

> All XLAL functions shall have a name beginning with `XLAL` followed by an uppercase letter.

> XLAL functions shall not call LAL functions.

> The return type of XLAL functions shall be one of: `int`, `CHAR`, `INT2`, `INT4`, or `INT8` (integer-type return XLAL functions); `REAL4` or `REAL8` (floating-point-type return XLAL functions); a pointer (pointer-type return XLAL functions); or no return type (type `void` return XLAL functions).

5.1 Four kinds of XLAL functions

XLAL functions will be one of four types based principally on their return type, though this is also largely determined by their functional nature. The way that the XLAL functions report an error through their return value depends on which type it is. In addition, all XLAL functions will report errors by setting an XLAL error number, `xlalErrno`, and invoking the XLAL error handler (described below). For each type of function there is a macro that will perform all of these tasks.

1. **XLAL functions that return an integer.** These are XLAL functions that return one of `CHAR`, `INT2`, `INT4`, `INT8`, or `int`.

   Simple XLAL functions will return type `int` that will either be `0` to indicate success or `-1` to indicate failure. However, sometimes it is useful to have an XLAL function that counts things (e.g., nodes in a linked list). For these functions it is useful for the count to be the return value. Therefore the rule for XLAL functions in this category is:

   > All XLAL functions that return an integer type shall return a negative result to indicate a failure. In addition, the `xlalErrno` shall be set to the appropriate error number and the XLAL error handler shall be invoked.

   This means that there cannot be an XLAL function that returns an unsigned integer type (including `size_t`).

   To report an error from this type of function, use the macro `XLAL_ERROR( func, errnum )` where `func` is the function name string and `errnum` is the XLAL error number (see below).

2. **XLAL functions that return a floating-point number.** These are XLAL functions that return either `REAL4` or `REAL8`.

   Such functions are quite useful for providing extended mathematical functions to do things such as compute the value of a distribution at a certain point, etc. The value returned must still be checked to see if there was an error. To flag an error, these functions should return a particular value that would be impossible to obtain. The value is given by the constants `XLAL_REAL4_FAIL_NAN` which has the same bit pattern as the 32 bit hexadecimal integer constant `0x7fc001a1`, or `XLAL_REAL8_FAIL_NAN` which has the same bit pattern as the 64 bit hexadecimal integer constant `0x7ff80000000001a1` respectively. These constants are known as "quiet" (as opposed to "signaling") NaN (not-a-number) values. However, owing to the `1a1` at the end of the hexadecimal representation, they are not likely to occur as a result of any calculation (e.g., `0.0/0.0`) as it is unlikely that any C library will use these particular NaN values. Thus these values are identifiable as failures arising from XLAL functions and represent impossible results.

   To summarize:

   > All XLAL functions that return a `REAL4` floating-point type shall return the `REAL4` floating-point constant `XLAL_REAL4_FAIL_NAN` to indicate a failure. All XLAL functions that return a `REAL8` floating-point type shall return the `REAL8` floating-point constant `XLAL_REAL8_FAIL_NAN` to indicate a failure. In addition, the `xlalErrno` shall be set to the appropriate error number and the XLAL error handler shall be invoked.

   To report an error from these types of functions, use one of the macros `XLAL_ERROR_REAL4( func, errnum )` or `XLAL_ERROR_REAL8( func, errnum )` where `func` is the function name string and `errnum` is the XLAL error number (see below). The result from a function call must be checked to see if one of these constants has been returned. This can be done with the macros `XLAL_IS_REAL4_FAIL_NAN(val)` and `XLAL_IS_REAL8_FAIL_NAN(val)`.

3. **XLAL functions that return a pointer.** These are often XLAL functions that are used to create structures, but can also be functions that return a pointer to the output structure. An example of the latter, imagine the function:

```
COMPLEX8 *XLALCOMPLEX8Add( COMPLEX8 *result, COMPLEX8 *val1, COMPLEX8 *val2 )
{
  if ( ! result || ! val1 || ! val2 ) /* NULL argument */
    XLAL_ERROR_NULL( "XLALCOMPLEX8Add", XLAL_EFAULT );
  result->re = val1->re + val2->re;
  result->im = val1->im + val2->im;
  return result;
}
```

   The `XLAL_ERROR_NULL( func, errnum )` macro prints out the function name `func`, sets the XLAL errno to `errnum` (in this case the error number is `XLAL_EFAULT`) and invokes the XLAL error handler (see below). It then returns `NULL`. All functions of this type will indicate an error by returning `NULL`:

   > All XLAL functions that return a pointer type shall return the result `NULL` to indicate a failure. In addition, the `xlalErrno` shall be set to the appropriate error number and the XLAL error handler shall be invoked.

   To report an error from this type of function, use the macro `XLAL_ERROR_NULL( func, errnum )` where `func` is the function name string and `errnum` is the XLAL error number (see below).

4. **XLAL functions that do not have a return value (return `void`.** These are functions that *really shouldn't* fail. In practice, they are almost always free-type functions that destroy memory created by the create-type functions of the previous function type. There is no way to return a success/failure flag via the return value so all success/failure information must be returned through the XLAL error number `xlalErrno`.

   > All XLAL functions that do not return a result (i.e., they return `void`) shall set the `xlalErrno` to the appropriate error number and shall invoke the XLAL error handler.

   To report an error from this type of function, use the macro `XLAL_ERROR_VOID( func, errnum )` where `func` is the function name string and `errnum` is the XLAL error number (see below).

5.2 XLAL error numbers and error handlers

XLAL functions use the modifiable lvalue `xlalErrno` (think of it as a global `int`-type variable) to codify the nature of a failure. It should not be used for any other purpose. The values that `xlalErrno` is allowed to have are controlled. It is quite analogous to the standard C `errno`.

To use `xlalErrno`, set it to zero (no error) before calling an XLAL function; call the function; and then check the value of `xlalErrno`. If it is non-zero, an error has occurred, and the value can be used to determine the nature of the error. The following table contains the XLAL return codes and error numbers. Note that `xlalErrno` should only be set to one of the error numbers (or zero if there is no error).

Code	Value	Meaning
Return codes (for XLAL functions that return `int`)		
`XLAL_SUCCESS`	0	Success
`XLAL_FAILURE`	−1	Failure
Error numbers		
Standard error numbers		
`XLAL_EIO`	5	I/O error
`XLAL_ENOMEM`	12	Memory allocation error
`XLAL_EFAULT`	14	Invalid pointer
`XLAL_EINVAL`	22	Invalid argument
`XLAL_EDOM`	33	Input domain error
`XLAL_ERANGE`	34	Output range error
Extended error numbers begin at 128		
Common error numbers for XLAL functions		
`XLAL_EFAILED`	128	Generic failure
`XLAL_EBADLEN`	129	Inconsistent or invalid vector length
Specific mathematical and numerical error numbers begin at 256		
IEEE floating point error numbers		
`XLAL_EFPINVAL`	256	Invalid floating point operation
`XLAL_EFPDIV0`	257	Division by zero floating point error
`XLAL_EFPOVRFLW`	258	Floating point overflow error
`XLAL_EFPUNDFLW`	259	Floating point underflow error
`XLAL_EFPINEXCT`	260	Floating point inexact error
Numerical algorithm error numbers		
`XLAL_EMAXITER`	261	Exceeded maximum number of iterations
`XLAL_EDIVERGE`	262	Series is diverging
`XLAL_ESING`	263	Apparent singularity detected
`XLAL_ETOL`	264	Failed to reach specified tolerance
`XLAL_ELOSS`	265	Loss of accuracy
Failure from within a function call: "or" error number with this		
`XLAL_EFUNC`	1024	Internal function call failed

Note that the last error number, `XLAL_EFUNC`, corresponds to a bit that can be set on the current error number (using a bitwise-or) to indicate that the this error occurred from within an internal function call, thereby preserving some information about the error.

In addition to setting `xlalErrno` a failure condition should also invoke the XLAL error handler `XLALErrorHandler`. This is a function pointer (actually it can be a macro that results in a function pointer) to a function; its type is

```
typedef void XLALErrorHandlerType( const char *func, const char *file, int line, int errnum );
```

Thus the error handler takes the name of the function from which it is invoked, `func`, the file name of the source, `file`, the line number where it is called, `line`, and the XLAL error number `errnum`. The default error handler, `XLALDefaultErrorHandler`, will print an error message when it is invoked. The user may set the error handler to a different error handler, e.g., one that aborts when a failure occurs. *However, the error handler should not be changed within a LAL or an XLAL function.* Replacing the error handler should always be done in the top-level program.

To assist in setting `xlalErrno` and invoking the error handler, the function

```
void XLALError( const char *func, const char *file, int line, int errnum );
```

is provided which will perform both of these tasks. The arguments are the same as those of the error handler. This function is called as part of the actions of the macros XLAL_ERROR, XLAL_ERROR_NULL, XLAL_ERROR_VOID, XLAL_ERROR_REAL4, and XLAL_ERROR_REAL8, all of which take two argument: a character string containing the name of the current function and the integer error number. These macros call XLALError with the filename given by __FILE__, the line number __LINE__ where the macro occurs, and with the function name and error number. They also return from the function with the appropriate failure return code (depending on which macro was used).

6 Documentation and unit tests

The conventions for these are not within the scope of this specification; they are described in the LAL Software Documentation. It is a good guide that every function with external linkage should have a unit test that can be run automatically to make sure it is (and continues to be) sane.

7 Other libraries required for LAL

LAL is not a stand-alone library. Two other libraries are required to build and use LAL. These are the "Fastest Fourier Transform in the West (version 3)" FFTW3 library (compiled in both single and double precision formats) and the "GNU Scientific Library" GSL library. The way these libraries are integrated into LAL is different.

The FFTW3 library is integrated by wrapping certain FFTW3 routines within LAL functions. Other LAL functions should then use these wrapping functions rather than make direct calls to the FFTW3 API. This is possible because only a few functions in FFTW3 are needed in LAL.

The GSL library provides many more functions than FFTW3. Some of these functions, e.g., those involving file I/O, are not suitable for use within LAL. However, the vast majority of the functions in GSL are useful. To facilitate their use within LAL, the macros CALLGSL(statement, status) and TRYGSL(statement, status) are provided. These macros wrap the statement statement within a set of code designed to (i) ensure the thread-safety and standard behavior of the error handler used by the GSL function call, and (ii) report any error conditions reported by the GSL function in the LAL status structure status.

8 Notable exceptions

There will be exceptions to (nearly) all of these rules. Exceptions are under the strict control of the LAL librarian.

Some parts of LAL must necessarily fail to conform to the above specifications. Clear examples include LALMalloc and their kin routines which do not use a LAL status structure and do not have void return (apart from LALFree) — indeed, since they must manage a heap, they are not really reentrant either. The LAL Librarian will endeavor to make LAL conform to the specification outlined in this document as much as is practical, but there are occasions when violations must be allowed. In such cases, the violations are under strict control of the LAL Librarian.

9 Beyond LAL... LALSupport, LALMetaIo, LALFrame

Of the LAL requirements, the most *functionally* limiting is the requirement that no I/O is allowed. This requirement is in place to insulate the bulk of the library from requirements about the nature of the system on which a program is being run. The philosophy is that since the library must always be integrated somehow into an executable program, assumptions about the system should be made by the program rather than by the library; hence the I/O should be contained within the program.

To assist a program with various I/O tasks and their integration with LAL, several libraries associated with LAL are provided. These libraries do not need to conform to all of the LAL standards; in particular, their purpose is to provide the I/O functions that are missing from LAL.

The LALSupport, LALMetaIo, and LALFrame libraries contain routines that need not conform to all of the LAL specifications; in particular, they contain routines that perform file I/O and/or require additional libraries. Routines from these libraries are intended to be used along with LAL routines, but LAL routines shall not call any routine from (or in any other way be dependent upon) these libraries.

Note that there are currently two official exchange data formats within the LSC: the XML-based "LIGO Lightweight" LIGOlw format, and the binary "Interferometric Gravitational Wave Detector Data Frame Format" or "Frame" format. Libraries with routines that are specialized for I/O with these formats are also available.

9.1 The LALSupport library

This library provides the basic file I/O routines that are used in conjunction with the LAL library. It is always built and installed along with the LAL library.

9.2 The LALMetaIo library

This library provides I/O routines that are used to read/write the LIGO lightweight data format. These routines use the METAIO library routines as their engine. This library is conditionally built and installed by LAL if the METAIO library is available.

9.3 The LALFrame library

This library provides I/O routines that are used to read/write the Frame data format. These routines use the FRAME library routines as their engine. This library is conditionally built and installed by LAL if the FRAME library is available.

A Language issues

The C99 standard specifies certain keywords and standard library functions. Only a subset of these are suitable for LAL functions. However, for maximum portability, one should avoid various extensions to the C99 keywords and functions. Here are some guidelines on writing portable LAL code.

A.1 Namespace requirements

Here is a list of keywords and reserved names. These should be avoided when choosing names of LAL variables, functions, etc. The LAL namespace will help avoid namespace collisions. The LAL namespace conventions are also given.

Keywords

The code should avoid any of the following keywords that are present in C++ as symbol names:

```
and           and_eq        asm           auto          bitand        bitor
bool          break         case          catch         char          class
compl         const         const_cast    continue      default       delete
do            double        dynamic_cast  else          enum          explicit
export        extern        false         float         for           friend
goto          if            inline        int           long          mutable
namespace     new           not           not_eq        operator      or
or_eq         private       protected     public        register      reinterpret_cast
return        short         signed        sizeof        static        static_cast
struct        switch        template      this          throw         true
try           typedef       typeid        typename      union         unsigned
using         virtual       void          volatile      wchar_t       while
xor           xor_eq
```

It is not a good idea to use `fortran` or `entry` either as these are sometimes reserved. Of these keywords, the ones that are C99 keywords are

```
_Bool           _Complex        _Imaginary      auto
break           case            char            const
continue        default         do              double
else            enum            extern          float
for             goto            if              inline
int             long            register        restrict
return          short           signed          sizeof
static          struct          switch          typedef
union           unsigned        void            volatile
while
```

Of these there is (almost) no need to use `auto`, `char`, `double`, `float`, `long`, `register`, `short`, `signed`, `unsigned`, or `volatile`.

Reserved names

According to the GNU C library, the following names are reserved (or may be reserved in the future) by the C library:

- All global functions or variables that begin with an underscore, e.g., `_whatever`, are reserved.
- All identifiers that begin with two underscores or with an underscore followed by an uppercase letter e.g., `__whatever`, `_Whatever`, are reserved.
- Names beginning with the capital `E` and followed by a digit or uppercase letter are reserved (for error codes).
- Names beginning with `is` or `to` and followed by a lowercase letter are reserved.
- Names beginning with `LC_` are reserved.
- Names of all existing mathematical functions but suffixed with either an `f` or an `l` are reserved.
- Names of all existing mathematical functions but suffixed with either an `f` or an `l` are reserved.
- Names beginning with `SIG` or `SIG_` and followed by an uppercase letter are reserved.
- Names beginning with `str`, `mem`, or `wcs` and followed by a lowercase letter are reserved.
- Names ending with `_t` are reserved.

Certain headers reserve names too. Since LAL is a library to be used by others, it is important to respect these when coding the interface.

- Names prefixed with `d_` are reserved in `dirent.h`.
- Names prefixed with `l_`, `F_`, `O_` and `S_` are reserved in `fcntl.h`.
- Names prefixed with `gr_` are reserved in `grp.h`.
- Names suffixed with `_MAX` are reserved in `limits.h`.
- Names prefixed with `pw_` are reserved in `pwd.h`.
- Names prefixed with `sa_` and `SA_` are reserved in `signal.h`.
- Names prefixed with `st_` and `S_` are reserved in `sys/stat.h`.
- Names prefixed with `tms_` are reserved in `sys/times.h`.
- Names prefixed with `c_`, `V`, `I`, `O`, `TC` and names prefixed with `B` followed by a digit are reserved in `termios.h`.

LAL namespace

The LAL namespace will assist in avoiding namespace conflicts. LAL reserves any name that is prefixed with `LAL`, `LAL_`, `XLAL`, `XLAL_`, `lal`, or `xlal` followed by an uppercase letter. In addition LAL reserves names that begin with `CHAR`, `UCHAR`, `INT2`, `INT4`, `INT8`, `UINT2`, `UINT4`, `UINT8`, `REAL4`, `REAL8`, `COMPLEX8`, `COMPLEX16`, and `LIGO` followed by an uppercase letter.

A.2 Allowed functions from standard C

C libraries often have various extensions from the C99 standard C library, but for portability purposes only those functions that are specified by the C99 standard should be used (note: LAL does require a *hosted* rather than *freestanding* environment). Also, many of these should *not* be used in LAL because they would cause the LAL functions to violate one of the above rules (e.g., would be used for file I/O, would cause a function to not be reenterant, etc.). For completeness, these are the allowed standard C functions. There are a few others that could be allowed, but are not recommended (e.g., `sprintf`, but `LALSprintf` is provided as a preferred alternative) and are not listed here for that reason. Also listed here are macros and types that are defined in these headers. If a function or macro or type is not somewhere on this list, you should probably not use it in a LAL function.

```
<stdio.h>
        sscanf
        EOF

<ctype.h>
        isalnum         isalpha         iscntrl         isdigit
        isgraph         islower         isprint         ispunct
        isspace         isupper         isxdigit
        tolower         toupper

<string.h>
        strcpy          strncpy         strcat          strncat
        strcmp          strncmp         strchr          strrchr
        strspn          strcspn         strpbrk         strstr
        strlen
        memcpy          memmove         memcmp          memchr
        memset
        NULL            size_t

<math.h>
        sin             cos             tan
        asin            acos            atan            atan2
        sinh            cosh            tanh
        exp             log             log10           pow
        sqrt            ceil            floor           fabs
        ldexp           frexp            modf           fmod
        HUGE_VAL

<stdlib.h>
        atof            atoi            atol
        strtod          strtol          strtoul
        bsearch         qsort
        abs             labs            div             ldiv
        NULL            size_t          div_t           ldiv_t

<errno.h>
        errno
        EDOM            ERANGE

<assert.h> NONE

<stdarg.h>
```

```
        va_start        va_arg          va_end

<setjmp.h> NONE

<signal.h> NONE

<time.h>
        difftime        mktime          strftime
        NULL            size_t          time_t          struct tm

<limits.h> NONE

<float.h> NONE
```

B LAL Datatypes

B.1 Primitive Datatypes

The primitive datatypes are defined in a separate header `LALAtomicDatatypes.h`, which is included by `LALDatatypes.h`. This is done in order to facilitate the interface between LAL and non-LAL modules. By including just `LALAtomicDatatypes.h`, a non-LAL module can ensure that it is using the same arithmetic standard as LAL, without being burdened by LAL's more specialized structures.

Primitive datatypes are those that conceptually store a single number or quantity. They include both the *atomic* datatypes and the complex datatypes.

Atomic Datatypes

Atomic LAL datatypes are platform-independent datatypes corresponding to the basic types in the C/C++ language. However, since the C/C++ types are not necessarily the same across platforms, the actual mapping between LAL and C/C++ datatypes may be different on different platforms. The following table lists the LAL atomic datatypes, their size and range, and the C/C++ datatype to which they *usually* correspond.

Type	Bits	Range	Usual C/C++ type
`CHAR`	8	`'\0'` to `'\255'`	`char`
`UCHAR`	8	`'\0'` to `'\255'`	`unsigned char`
`INT2`	16	-2^{-15} to $2^{15}-1$	`short`
`INT4`	32	-2^{-31} to $2^{31}-1$	`int` or `long`
`INT8`	64	-2^{-63} to $2^{63}-1$	`long long`
`UINT2`	16	0 to $2^{16}-1$	`unsigned short`
`UINT4`	32	0 to $2^{32}-1$	`unsigned int` or `long`
`UINT8`	64	0 to $2^{64}-1$	`unsigned long long`
`REAL4`	32	-3.4×10^{38} to 3.4×10^{38}	`float`
`REAL8`	64	-1.8×10^{308} to 1.8×10^{308}	`double`

The unsigned character and integer datatypes store their values according to the usual binary system. For signed characters and integers, setting the most-significant bit indicates that the number formed from the remaining bits should be added to the lower value of the range. The `REAL4` and `REAL8` datatypes should store values according to the IEEE Standard 754 for Binary Floating-Point Arithmetic, which gives them the following precisions and dynamic ranges:

	`REAL4`	`REAL8`
Minimum positive subnormal	1.4×10^{-45}	4.9×10^{-324}
Minimum positive normal	1.2×10^{-38}	2.2×10^{-308}
Maximum finite normal	3.4×10^{38}	1.8×10^{308}
Minimum fractional difference	6.0×10^{-8}	1.1×10^{-16}
Significant decimal digits	6–9	15–17

The minimum positive subnormal is the smallest positive representable number. The minimum positive normal is the smallest positive number that can be represented with full precision; that is, one whose mantissa lies in the range [0.5,1). The maximum finite normal is the largest representable number other than the reserved code for $+\infty$. The minimum fractional difference is the smallest fractional difference between consecutive representable numbers, or *half* the difference between 1 and the next representable number. Significant decimal digits gives the number of decimal digits used to represent the binary number in decimal notation: the first is the maximum number of digits that are guaranteed not to change upon conversion to binary, the second is the number of digits required to represent a unique binary quantity.

Complex datatypes

LAL represents complex numbers as structures with two floating-point fields, storing the real and imaginary parts. These are considered primitive datatypes (rather than aggregate or structured datatypes) because they conceptually represent a single number. Furthermore, atomic and complex datatypes are treated equivalently by LAL aggregate and structured datatypes.

`COMPLEX8`: This structure stores a single-precision complex number in 8 bytes of memory. The fields are:

– **`REAL4 re`** The real part.

– **`REAL4 im`** The imaginary part.

`COMPLEX16`: This structure stores a double-precision complex number in 16 bytes of memory. The fields are:

– **`REAL8 re`** The real part.

– **`REAL8 im`** The imaginary part.

B.2 Aggregate datatypes

These datatypes store arbitrarily large sets or collections of primitive datatypes. At this level there is no physical interpretation assigned to the objects (such as names or units); the aggregate datatypes simply collect and arrange the primitive datatypes. The following types of aggregate datatypes are defines: vectors, arrays, sequences, vector sequences, and array sequences.

⟨*datatype*⟩**`Vector`:** This structure stores an ordered set of n elements of type ⟨*datatype*⟩, which can be any primitive datatype. The data are to be interpreted as being a point in an n-dimensional vector space. The fields are:

– **`UINT4 length`** The number of data n.

– ⟨*datatype*⟩ **`*data`** Pointer to the data array. The data are stored sequentially as `data[`$0,\ldots,n-1$`]`.

⟨*datatype*⟩**`Array`:** This structure stores a set of elements of type ⟨*datatype*⟩, which can be any primitive datatype, arranged as an m-dimensional array. That is, each element can be thought of as having m indices, $\mathsf{A}_{i_0\cdots i_{m-1}}$, where each index i_k runs over its own range $0,\ldots,n_k-1$. The total number of elements is then $N = n_0 \times \cdots \times n_{m-1}$. In memory the array is "flattened" so that the elements are stored sequentially in a contiguous block. The fields are:

– **`UINT4Vector *dimLength`** Pointer to a vector of length m, storing the index ranges $(n_0,\ldots,n_{m-1})$.

– ⟨*datatype*⟩ **`*data`** Pointer to a pointer to the data array. The data element $\mathsf{A}_{i_0\cdots i_{m-1}}$ is stored as `data[`$i_{m-1}+n_{m-2}\times(i_{m-2}+n_{m-3}\times(\cdots(i_1+n_0\times i_0)\cdots))$`]`; that is, the index of `data[]` runs over the entire range of an index i_{k+1} before incrementing i_k.

⟨*datatype*⟩**`Sequence`:** This structure stores an ordered set of l elements of type ⟨*datatype*⟩, which can be any primitive datatype. It is identical to ⟨*datatype*⟩`Vector` and is retained for historical purposes only.

⟨*datatype*⟩**VectorSequence:** This structure stores an ordered set of l elements of type ⟨*datatype*⟩`Vector`, where ⟨*datatype*⟩ can be any primitive datatype. Mathematically the sequence can be written as $\{\vec{v}^{(0)}, \ldots, \vec{v}^{(l-1)}\}$, where each element $\vec{v}^{(j)} = (v_0^{(j)}, \ldots, v_{n-1}^{(i)})$ is a vector of length n. In memory the elements are "flattened"; that is, they are stored sequentially in a contiguous block of memory. The fields are:

– **UINT4 length** The number of vectors l.

– **UINT4 vectorLength** The length n of each vector.

– ⟨*datatype*⟩ ***data** Pointer to the data array. The data element $v_i^{(j)}$ is stored as `data[`$j \times n + i$`]`; that is, the index of `data[]` runs over the internal index of each vector element before incrementing to the next vector element.

⟨*datatype*⟩**ArraySequence:** This structure stores an ordered set of l elements of type ⟨*datatype*⟩`Array`, where ⟨*datatype*⟩ can be any primitive datatype. The indexing of an array sequence can get quite complicated; it helps to read first the documentation for data arrays, above. Mathematically the data can be written as a set $\{\mathsf{A}^{(j)}_{i_0 \cdots i_{m-1}}$, where the sequence number j runs from 0 to $l-1$, and each array index i_k runs over its own range $0, \ldots, n_k - 1$. The total number of data in a given array element is then $N = n_0 \times \cdots \times n_{m-1}$, and the total number of data in the sequence is $N \times l$. In memory the array is "flattened" so that the elements are stored sequentially in a contiguous block. The fields are:

– **UINT4 length** The number l of array elements in the sequence.

– **UINT4 arrayDim** The number of data N (*not* the number of indecies m) in each array element of the sequence.

– **UINT4Vector *dimLength** Pointer to a vector of length m, storing the index ranges $(n_0, \ldots, n_{m-1})$.

– ⟨*datatype*⟩ ***data** Pointer to the data. The element $\mathsf{A}^{(j)}_{i_0 \cdots i_{m-1}}$ is stored as `data[`$j \times N + i_{m-1} + n_{m-2} \times (i_{m-2} + n_{m-3} \times (\cdots (i_1 + n_0 \times i_0) \cdots))$`]`; that is, the index of `data[]` runs over the internal indecies of each array element before incrementing to the next array element.

B.3 Structured datatypes

These datatypes embed primitive and aggregate datatypes inside structures that define their physical meaning. Most of these structures are wrappers for aggregate datatypes that store a physical quantity as a function of time or frequency. Other structures store specific physical information, such as the GPS time, or the factored response function of a filter.

LIGOTimeGPS: This structure stores the time, to nanosecond precision, synchronized to the Global Positioning System time reference. The zero time for the GPS standard is the moment of midnight beginning January 6, 1980, UTC. The `LIGOTimeGPS` structure can represent times up to 68 years on either side of this epoch. (Note that this is better than an equivalently-sized `REAL8` representation of time, which can maintain nanosecond precision only for times within 104 days of its reference point. However, the `REAL8` representation does allow one to cover arbitrarily long timescales at correspondingly lower precision.) The fields are:

– **INT4 gpsSeconds** The number of seconds since the GPS reference time.

– **INT4 gpsNanoSeconds** The number of nanoseconds since the last GPS second.

LALUnit: This structure stores units in the mksA system (plus Kelvin, Strain, and ADC Count). It also stores an overall power-of-ten scaling factor. The fields are:

– **INT2 powerOfTen** The power p of ten scaling factor.

– **INT2 unitNumerator[LALNumUnits]** Array of unit numerators, N_i, $i = 0 \ldots$ LALNumUnits $- 1$.

– **INT2 unitDenominatorMinusOne[LALNumUnits]** Array of unit denominators-minus-one, D_i, $i = 0 \ldots$ LALNumUnits$-$ 1.

Thus, the units are given by

$$10^p \times \mathrm{m}^{N_0/(1+D_0)} \times \mathrm{kg}^{N_1/(1+D_1)} \times \mathrm{s}^{N_2/(1+D_2)} \times \mathrm{A}^{N_3/(1+D_3)} \times \mathrm{K}^{N_4/(1+D_4)} \times \mathrm{strain}^{N_5/(1+D_5)} \times \mathrm{count}^{N_6/(1+D_6)} \quad (1)$$

The indexes of the units can be specified using the constants `LALUnitIndexMeter`, `LALUnitIndexKiloGram`, `LALUnitIndexSecond`, `LALUnitIndexAmpere`, `LALUnitIndexKelvin`, `LALUnitIndexStrain`, `LALUnitIndexADC` while `LALNumUnits` is the total number of units.

⟨*datatype*⟩**`TimeSeries`:** This structure represents a sequence of data of type ⟨*datatype*⟩ (where ⟨*datatype*⟩ can be any primitive datatype), sampled over uniform time intervals $t_0, t_0 + \Delta t, \ldots, t_0 + l\Delta t$. Essentially this is a ⟨*datatype*⟩`Sequence` with extra fields defining the sample times and the type of data being sampled. The raw data may also have been *heterodyned*; that is, multiplied by a sinusoid of some frequency f_0, low-pass filtered, and resampled, in order to extract the behavior in a small bandwidth about f_0. The fields are:

– **`CHAR name[LALNameLength]`** The name of the data series (i.e. the type of data being sampled).

– **`LIGOTimeGPS epoch`** The start time t_0 of the data series.

– **`REAL8 deltaT`** The sampling interval Δt, in seconds.

– **`REAL8 f0`** The heterodyning frequency f_0, in hertz.

– **`LALUnit sampleUnits`** The physical units of the quantity being sampled.

– ⟨*datatype*⟩**`Sequence *data`** The sequence of sampled data.

⟨*datatype*⟩**`FrequencySeries`:** This structure represents a frequency spectrum of data of type ⟨*datatype*⟩ (where ⟨*datatype*⟩ can be any primitive datatype), sampled over uniform frequency intervals $f_0, f_0 + \Delta f, \ldots, f_0 + l\Delta f$. Essentially this is a ⟨*datatype*⟩`Sequence` with extra fields defining the sample frequencies, the timestamp of the spectrum, and the type of data being sampled. The fields are:

– **`CHAR name[LALNameLength]`** The name of the data series (i.e. the type of data being sampled).

– **`LIGOTimeGPS epoch`** The start time of the *time* series from which the spectrum was calculated.

– **`REAL8 f0`** The lowest frequency f_0 being sampled, in hertz.

– **`REAL8 deltaF`** The frequency sampling interval Δf, in hertz.

– **`LALUnit sampleUnits`** The physical units of the quantity being sampled.

– ⟨*datatype*⟩**`Sequence *data`** The sequence of sampled data.

⟨*datatype*⟩**`ZPGFilter`:** This structure stores the complex frequency response of a filter or transfer function in a factored form, where ⟨*datatype*⟩ can be either `COMPLEX8` or `COMPLEX16`. One defines a (dimensionless) complex frequency variable $\zeta(f\Delta t)$, where Δt is the time sampling interval of the data to which the filter will be applied (in the case of a digital filter), or some other reference timescale (in the case of an analog filter). The complex response function can then be given (or approximated) as $H(f) = g \times \prod_k (\zeta - z_k) / \prod_l (\zeta - p_l)$, where z_k are the complex *zeros*, p_l are the complex *poles*, and g is the complex *gain* of the response function. Some common complex frequency representations are the z-plane representation $\zeta(f\Delta t) = \exp(2\pi i f \Delta t)$, which maps the Nyquist interval $f \in [0, 1/2\Delta t)$ onto the upper-half unit circle in ζ, and the w-plane representation $\zeta(f\Delta t) = \tan(\pi f \Delta t)$, which maps the Nyquist interval onto the positive real axis in ζ. The fields of ⟨*datatype*⟩`ZPGFilter` are:

– **`CHAR name[LALNameLength]`** The name of the filter or transfer function. This should also mention its complex frequency representation.

– **`REAL8 deltaT`** The sampling time or reference timescale Δt for the filter, in seconds. If zero, it will be treated as being equal to the sampling interval of the data being filtered.

– ⟨*datatype*⟩**`Vector *zeros`** Pointer to a vector storing the zeros z_k of the filter.

– ⟨*datatype*⟩**`Vector *poles`** Pointer to a vector storing the poles p_k of the filter.

– ⟨*datatype*⟩ **`gain`** The gain g of the filter.

B.4 The LAL universal status structure `LALStatus`

This structure is the means by which LAL functions report their success or failure; it provides a useful mechanism for tracking progress and errors through nested function calls. The error reporting structure is a linked list of `LALStatus` structures, with each node corresponding to a given function in the current calling sequence. When a function terminates successfully, its node is dropped from the list. If a function encounters an error, it must still return control to the calling routine, reporting the error through its `LALStatus`. The calling routine must either deal with the error (pruning the linked list if it succeeds), or else return an error itself. A fatal error will thus return a linked list of `LALStatus` structures to the top-level routine, where the tail of the list identifies the source of the error, and the intermediate nodes identify the sequence of nested function calls that led to the error. The fields of the `LALStatus` are as follows:

– **`INT4 statusCode`** A numerical code identifying the type of error, or 0 for nominal status.

– **`const CHAR *statusDescription`** A description of the current status or error.

– **`volatile const CHAR *Id`** The RCS ID string of the source file of the current function.

– **`const CHAR *function`** The name of the current function.

– **`const CHAR *file`** The name of the source file of the current function.

– **`INT4 line`** The line number in the source file where the current `statusCode` was set.

– **`LALStatus *statusPtr`** Pointer to the next node in the list; `NULL` if this function is not reporting a subroutine error.

– **`INT4 level`** The current level in the nested calling sequence.

C The `LALStatus` structure

LAL routines store their current execution status in a linked list of structures of type `LALStatus`, with each node in the list representing a subroutine in the current calling sequence. The `LALStatus` structure is described in Sec. B.4 of the header `LALDatatypes.h`, but for completeness, we explain its fields below:

– **`INT4 statusCode`** A code indicating the exit status of a function. 0 represents a normal exit. Negative values are reserved for certain standard error types. The authors of individual functions should assign positive values to the various ways in which their code can fail.

– **`const CHAR *statusDescription`** An explanatory string corresponding to the numerical status code.

– **`volatile const CHAR *Id`** A character string identifying the source file and version number of the function being reported on.

– **`const CHAR *function`** The name of the function.

– **`const CHAR *file`** The file name of the `.c` file containing the function code.

– **`INT4 line`** The line number in the `.c` file of the instruction where any error was reported.

– **`LALStatus *statusPtr`** A recursive pointer to another status pointer. This structure is used to report an error in a subroutine of the current function. Thus if an error occurs in a deeply-nested routine, the status structure returned to the main program will be the head of a linked list of status structures, one for each nested level, with the tail structure reporting the actual error that caused the overlying routines to fail.

– **`INT4 level`** The nested-function level where any error was reported.

In almost all circumstances the programmer will *not* have to access this structure directly, relying instead on the macros defined in the header `LALStatusMacros.h`. The exception is the `statusCode` field, which the programmer may want to query directly.

The `statusCode` field is set to a nonzero value any time an error condition arises that would lead to abnormal termination of the current function. Programmers can assign positive error codes to the various types of error that may be encountered in their routines. Additionally, the following following status codes are reserved to report certain standard conditions:

Code	Message	Explanation
0		Nominal execution; the function returned successfully.
-1	Recursive error	The function aborted due to failure of a subroutine.
-2	INITSTATUS: non-null status pointer	The status structure passed to the function had a non-NULL statusPtr field, which blocks the function from calling subroutines (it is symptomatic of something screwy going on in the calling routine).
-4	ATTATCHSTATUSPTR: memory allocation error	The function was unable to allocate a statusPtr field to pass down to a subroutine.
-8	DETATCHSTATUSPTR: null status pointer	The statusPtr field could not be deallocated at the end of all subroutine calls; one of the subroutines must have lost it or set it to NULL.

D The lalDebugLevel

The lalDebugLevel is a global variable, set at runtime, that determines how much and what kind of debugging information will be reported. It is declared as an extern int in the header LALStatusMacros.h, and is therefore accessible in any standard LAL module that includes this header. Note, however, that it is declared to be of the C type int, which is usually but not always a 32-bit integer (on some systems it may only be 16 bits).

The value of lalDebugLevel should be thought of not as a number, but as a *bit mask*, wherein each bit in the binary representation turns on or off a specific type of status reporting. At present, there are five types of status reporting, each associated with a bit in lalDebugLevel.

Error messages tell the operator that a computation has terminated abnormally, and has failed to produce an acceptable result. Normally this is associated with assigning a non-zero statusCode; an error message is printed automatically whenever a function exits with non-zero statusCode.

Warning messages tell the user that a computation is working, but with unusual behavior that might indicate an unreliable or meaningless result. Warnings do not normally result in a non-zero statusCode.

Information messages tell the operator that the computation is proceeding as expected, and simply provide additional information about its progress.

Tracing messages are printed automatically a subroutine is called or returned; they simply track the current sequence of function calls.

Memory information messages are a special type of information message; they tell the operator when and how much memory is allocated or freed from the memory heap.

The module LALError.c defines functions for printing each of these types of status message. Each type of message is turned on by setting the corresponding bit in lalDebugLevel to 1, and is suppressed by setting the bit to 0. This header file #defines flags with numerical values designed to switch on the appropriate bits. Combinations of bits can be switched on by combining these flags using the bitwise-*or* operator, |. The flags are defined as follows:

Flag	Octal	Decimal	Meaning
Primitive flags			
LALNDEBUG	000000	0	No debugging or status messages
LALERROR	000001	1	Turn on error messages
LALWARNING	000002	2	Turn on warning messages
LALINFO	000004	4	Turn on info messages
LALTRACE	000010	8	Turn on tracing messages
LALMEMINFO	000020	16	Turn on memory messages
LALNMEMDBG	000040	32	Turn off all memory debugging
LALNMEMPAD	000100	64	Turn off memory padding
LALNMEMTRK	000200	128	Turn off memory tracking
LALMEMDBG	040000	16384	Turn on memory debugging without messages
Combination flags			
LALMSGLVL1	000001	1	Error messages only
LALMSGLVL2	000003	3	Error and warning messages
LALMSGLVL3	000007	7	Error, warning, and info messages
LALMEMTRACE	000030	24	Memory and tracing messages
LALALLDBG	077437	32543	All messages and debugging

The most significant bit of lalDebugLevel has a special meaning in that it is not associated with any type of status message. However, certain pieces of debugging or error-tracking code — such as the memory leak detection code in LALMalloc.c — do not write status messages and are not associated with a lalDebugLevel bit; instead, these pieces of code are turned on for *any* nonzero value of lalDebugLevel, unless the LALNMEMDBG bit is set. Switching on only the most significant bit with LALMEMDBG activates this code without turning on any other error reporting.

E LAL Constants

E.1 Mathematical Constants

The following constants define the precision and range of floating-point arithmetic in LAL. They are taken from the IEEE standard 754 for binary arithmetic. All numbers are dimensionless.

Name	Value	Description
LAL_REAL4_MANT	24	Bits in REAL4 mantissa
LAL_REAL4_MAX	$3.40282347 \times 10^{38}$	Largest REAL4
LAL_REAL4_MIN	$1.17549435 \times 10^{-38}$	Smallest positive REAL4
LAL_REAL4_EPS	$1.19209290 \times 10^{-7}$	$2^{-(\text{LAL\_REAL4\_MANT}-1)}$
LAL_REAL8_MANT	53	Bits in REAL8 mantissa
LAL_REAL8_MAX	$1.7976931348623157 \times 10^{308}$	Largest REAL8
LAL_REAL8_MIN	$2.2250738585072014 \times 10^{-308}$	Smallest positive REAL8
LAL_REAL8_EPS	$2.2204460492503131 \times 10^{-16}$	$2^{-(\text{LAL\_REAL8\_MANT}-1)}$

LAL_REAL4_EPS and LAL_REAL8_EPS can be thought of as the difference between 1 and the next representable REAL4 or REAL8 number.

The following are fundamental mathematical constants. They are mostly taken from the GNU C math.h header (with the exception of LAL_TWOPI, which was computed using Maple). All numbers are dimensionless.

Name	Value	Expression
LAL_E	2.7182818284590452353602874713526625	e
LAL_LOG2E	1.4426950408889634073599246810018922	$\log_2 e$
LAL_LOG10E	0.4342944819032518276511289189166051	$\log_{10} e$
LAL_LN2	0.6931471805599453094172321214581766	$\log_e 2$
LAL_LN10	2.3025850929940456840179914546843642	$\log_e 10$
LAL_SQRT2	1.4142135623730950488016887242096981	$\sqrt{2}$
LAL_SQRT1_2	0.7071067811865475244008443621048490	$1/\sqrt{2}$
LAL_GAMMA	0.5772156649015328606065120900824024	γ
LAL_PI	3.1415926535897932384626433832795029	π
LAL_TWOPI	6.2831853071795864769252867665590058	2π
LAL_PI_2	1.5707963267948966192313216916397514	$\pi/2$
LAL_PI_4	0.7853981633974483096156608458198757	$\pi/4$
LAL_1_PI	0.3183098861837906715377675267450287	$1/\pi$
LAL_2_PI	0.6366197723675813430755350534900574	$2/\pi$
LAL_2_SQRTPI	1.1283791670955125738961589031215452	$2/\sqrt{\pi}$
LAL_PI_180	$1.7453292519943295769236907684886127\times10^{-2}$	$\pi/180$
LAL_180_PI	57.295779513082320876798154814105170	$180/\pi$

E.2 Physical Constants

The following physical constants are defined to have exact values. The values of c and g are taken from [2], p_{atm} is from [3], while ϵ_0 and μ_0 are computed from c using exact formulae. They are given in the SI units shown.

Name	Value	Description
LAL_C_SI	$299\,792\,458\,\mathrm{m\,s^{-1}}$	Speed of light c in free space
LAL_EPSILON0_SI	$8.8541878176203898505365630317107503\times10^{-12}\,\mathrm{C^2N^{-1}m^{-2}}$	Permittivity ϵ_0 of free space
LAL_MU0_SI	$1.2566370614359172953850573533118012\times10^{-6}\,\mathrm{N\,A^{-2}}$	Permeability μ_0 of free space
LAL_GEARTH_SI	$9.80665\,\mathrm{m\,s^{-2}}$	Standard gravity g
LAL_PATM_SI	$101\,325\,\mathrm{Pa}$	Standard atmospheric pressure p_{atm}

The following are measured fundamental physical constants, with values given in [2]. When not dimensionless, they are given in the SI units shown.

Name	Value	Description
LAL_G_SI	$6.67259 \times 10^{-11}\,\mathrm{N\,m^2kg^{-2}}$	Gravitational constant G
LAL_H_SI	$6.6260755 \times 10^{-34}\,\mathrm{J\,s}$	Planck constant h
LAL_HBAR_SI	$1.05457266 \times 10^{-34}\,\mathrm{J\,s}$	Reduced Planck constant $\hbar$
LAL_MPL_SI	$2.17671 \times 10^{-8}\,\mathrm{kg}$	Planck mass
LAL_LPL_SI	$1.61605 \times 10^{-35}\,\mathrm{m}$	Planck length
LAL_TPL_SI	$5.39056 \times 10^{-44}\,\mathrm{s}$	Planck time
LAL_K_SI	$1.380658 \times 10^{-23}\,\mathrm{J\,K^{-1}}$	Boltzmann constant k
LAL_R_SI	$8.314511\,\mathrm{J\,K^{-1}}$	Ideal gas constant R
LAL_MOL	6.0221367×10^{23}	Avogadro constant
LAL_BWIEN_SI	$2.897756 \times 10^{-3}\,\mathrm{m\,K}$	Wien displacement law constant b
LAL_SIGMA_SI	$5.67051 \times 10^{-8}\,\mathrm{W\,m^{-2}K^{-4}}$	Stefan-Boltzmann constant σ
LAL_AMU_SI	$1.6605402 \times 10^{-27}\,\mathrm{kg}$	Atomic mass unit
LAL_MP_SI	$1.6726231 \times 10^{-27}\,\mathrm{kg}$	Proton mass
LAL_ME_SI	$9.1093897 \times 10^{-31}\,\mathrm{kg}$	Electron mass
LAL_QP_SI	$1.60217733 \times 10^{-19}\,\mathrm{C}$	Proton charge
LAL_ALPHA	$7.297354677 \times 10^{-3}$	Fine structure constant
LAL_RE_SI	$2.81794092 \times 10^{-15}\,\mathrm{m}$	Classical electron radius r_e
LAL_LAMBDAE_SI	$3.86159323 \times 10^{-13}\,\mathrm{m}$	Electron Compton wavelength λ_e
LAL_AB_SI	$5.29177249 \times 10^{-11}\,\mathrm{m}$	Bohr radius a
LAL_MUB_SI	$9.27401543 \times 10^{-24}\,\mathrm{J\,T^{-1}}$	Bohr magneton μ_B
LAL_MUN_SI	$5.05078658 \times 10^{-27}\,\mathrm{J\,T^{-1}}$	Nuclear magneton μ_N

E.3 Astrophysical Parameters

The following parameters are derived from measured properties of the Earth and Sun. The values are taken from [2], except for the obliquity of the ecliptic plane and the eccentricity of Earth's orbit, which are taken from [3]. All values are given in the SI units shown.

Name	Value	Description
LAL_REARTH_SI	$6.378140 \times 10^{6}\,\mathrm{m}$	Earth equatorial radius
LAL_AWGS84_SI	$6.378137 \times 10^{6}\,\mathrm{m}$	Semimajor axis of WGS-84 Reference Ellipsoid
LAL_BWGS84_SI	$6.356752314 \times 10^{6}\,\mathrm{m}$	Semiminor axis of WGS-84 Reference Ellipsoid
LAL_MEARTH_SI	$5.97370 \times 10^{24}\,\mathrm{kg}$	Earth mass
LAL_IEARTH	$0.409092804\,\mathrm{rad}$	Obliquity of the ecliptic (2000)
LAL_EEARTH	0.0167	Earth orbital eccentricity
LAL_RSUN_SI	$6.960 \times 10^{8}\,\mathrm{m}$	Solar equatorial radius
LAL_MSUN_SI	$1.98892 \times 10^{30}\,\mathrm{kg}$	Solar mass
LAL_MRSUN_SI	$1.47662504 \times 10^{3}\,\mathrm{m}$	Geometrized solar mass (length)
LAL_MTSUN_SI	$4.92549095 \times 10^{-6}\,\mathrm{s}$	Geometrized solar mass (time)
LAL_LSUN_SI	$3.846 \times 10^{26}\,\mathrm{W}$	Solar luminosity
LAL_AU_SI	$1.4959787066 \times 10^{11}\,\mathrm{m}$	Astronomical unit
LAL_PC_SI	$3.0856775807 \times 10^{16}\,\mathrm{m}$	Parsec
LAL_YRTROP_SI	$31\,556\,925.2\,\mathrm{s}$	Tropical year (1994)
LAL_YRSID_SI	$31\,558\,149.8\,\mathrm{s}$	Sidereal year (1994)
LAL_DAYSID_SI	$86\,164.09053\,\mathrm{s}$	Mean sidereal day
LAL_LYR_SI	$9.46052817 \times 10^{15}\,\mathrm{m}$	$c\times$tropical year (1994)

The following cosmological parameters are derived from measurements of the Hubble expansion rate and of the cosmic background radiation (CBR). Data are taken from [2]. In what follows, the normalized Hubble constant h_0 is equal to the actual Hubble constant H_0 divided by $\langle H \rangle = 100\,\mathrm{km\,s^{-1}Mpc^{-1}}$. Thus the Hubble constant can be written as:

$$H_0 = \langle H \rangle h_0 \; .$$

Similarly, the critical energy density ρ_c required for spatial flatness is given by:

$$\rho_c = \langle \rho \rangle h_0^2 \; .$$

Current estimates give h_0 a value of around 0.65, which is what is assumed below. All values are in the SI units shown.

Name	Value	Description
LAL_H0_SI	$2 \times 10^{-18}\,\mathrm{s}^{-1}$	Approx. Hubble constant H_0
LAL_H0FAC_SI	$3.2407792903 \times 10^{-18}\,\mathrm{s}^{-1}$	H_0/h_0
LAL_RHOC_SI	$7 \times 10^{-10}\,\mathrm{J\,m}^{-3}$	Approx. critical energy density ρ_c
LAL_RHOCFAC_SI	$1.68860 \times 10^{-9}\,\mathrm{J\,m}^{-3}$	ρ_c/h_0^2
LAL_TCBR_SI	2.726K	CBR temperature
LAL_VCBR_SI	$3.695 \times 10^{5}\,\mathrm{m\,s}^{-1}$	Solar velocity with respect to CBR
LAL_RHOCBR_SI	$4.177 \times 10^{-14}\,\mathrm{J\,m}^{-3}$	Energy density of CBR
LAL_NCBR_SI	$4.109 \times 10^{8}\,\mathrm{m}^{-3}$	Number density of CBR photons
LAL_SCBR_SI	$3.993 \times 10^{-14}\,\mathrm{J\,K}^{-1}\mathrm{m}^{-3}$	Entropy density of CBR

References

[1] LAL Software Documentation, `http://www.lsc-group.phys.uwm.edu/lal/lsd.pdf`

[2] Particle Data Group, R. M. Barnett et al., Phys. Rev. D**54**, 1 (1996)

[3] K. R. Lang, *Astrophysical Data: Planets and Stars*. Springer-Verlag, New York (1992)