The LSC Algorithm Library (LAL)
仕様書のレビュー

（Review of LIGO Scientific Collaboration Algorithm Library Specification and Style Guide）

Hirotaka Takahashi

Yamanashi Eiwa College

Documents

- LIGO-T04XXXX-A (2005/12/17)

https://www.lsc-group.phys.uwm.edu/daswg/projects/lal/lalspec.pdf

- LIGO-T990030-v2 (2010/03/25)

https://dcc.ligo.org/public/0010/T990030/002/T990030-v2.pdf

- LAL Software Documentation (lsd-4.0)

http://www.lsc-group.phys.uwm.edu/lal/lsd.pdf

INTRODUCTION

Purpose and Goal of the LAL software specification

- The LSC Algorithm Library (LAL) is a library of routines for use in gravitational wave data analysis.
  - The defining purpose of this document (LIGO-T990030-v2) 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 and the routines can easily be used by programs written in other languages (C++, Fortran, Python, etc.).
  - The rules in this specification may need to be periodically reexamined. Therefore, this is a living document. Librarian will amend this document as needed. Significant changes to the document will be made in consultation with the LSC Software Change Control Board.

Elements of the library specification

- Coding style :
  - In order to establish a library and to maintain a uniform look-and-feel.
- Function requirements :
  - In order to maintain portability and to establish a standard Application Programming Interface (API).
- Standard data structures, macros, and functions :
  - In order to assist in providing a common set of tools for developers to promote and collaborative development.

LAL and XLAL interfaces

- LAL functions tended to be large and monolithic, and often a particular “routine” was re-written many times in-line in many different functions.

- Instead, this specification introduces a second, parallel interface, called the XLAL interface, specifically for writing small, light-weight, helper routines.

CODING STYLE GUIDELINES

Coding style guidelines

- Atomic data types
  - LAL routines should use the LAL-specific atomic data types .
- Names of functions, variables, etc.
  - These rules are to define a standard namespace scheme. They apply to all functions with external linkage as well as types, macros, etc., in header files.
- Header and source file conventions
  - The LAL API is defined by the *installed* header files.
- Language requirements
  - LAL code should all be in “clean C,” i.e., that language that is a subset of both C and C++.
  - 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.
- Filename conventions
  - LAL has a rigid directory structure.

Atomic data types

- LAL routines should use the LAL-specific atomic data types :

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`

Names of functions, variables, etc. (1)

- These rules are to define a standard namespace scheme. They apply to all functions 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. （キャメルケース方式：アルファベットで複合語やフレーズを表記する際、各単語や要素語の先頭の文字を大文字で表記する手法）
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.

Names of functions, variables, etc. (2)

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`.

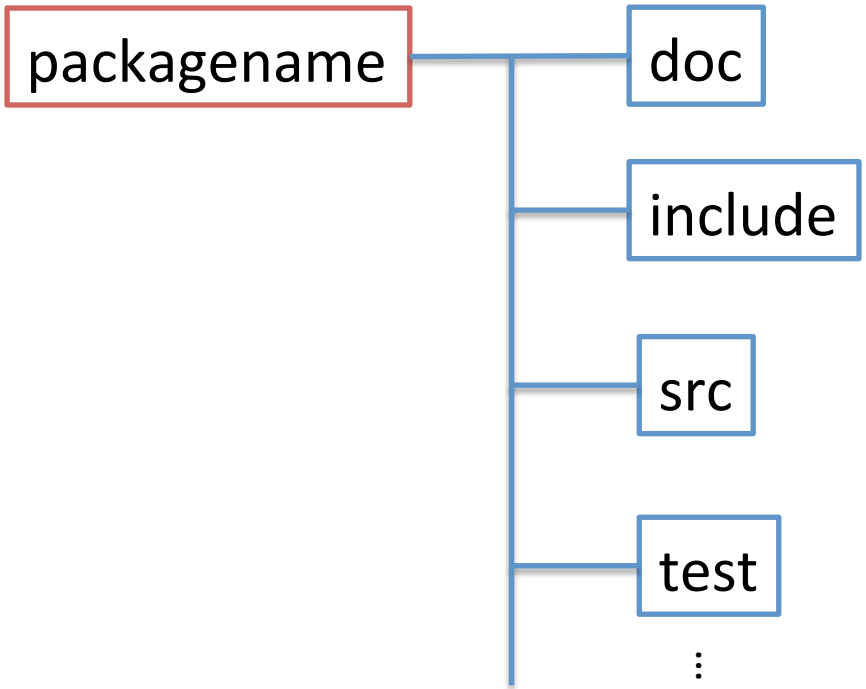
Header and source file conventions

- The LAL API is defined by the *installed* header files.
  - 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.
  - It is important that all source files (header files and the C source files) contain the Revision Control System (RCS) ID information.

```
#include <lal/LALRCSID.h> /* if no other LAL header has been included */
NRCSID( LALTHISHEADERH, "$Id: lalspec.tex,v 1.12 2005/05/03 02:37:30 patrick Exp $" );
```

Filename conventions

- LAL has a rigid directory structure :
  - LAL is composed of directories called packages whose names consist of entirely lower-case letters with no underscores.



Source files within these directories will be named with StudlyCaps.

COMMON RULES
FOR BOTH THE LAL AND THE XLAL FUNCTIONS

Common rules for both the LAL and the XLAL functions (1)

- Function arguments :

  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 (`...`).

- Functions should not have any dependence on system environment :

  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.

- Memory management :

  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.

Common rules for both the LAL and the XLAL functions (2)

- Functions must be reentrant and thread-safe :

> 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.

- Functions should always return control to the calling program :

> 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.

RULES FOR LAL FUNCTIONS

Rules for LAL functions (1)

- All LAL functions shall have names that begin with LAL and use StudlyCaps.
- All LAL functions must return void.
- All LAL functions have as their first argument a pointer to a LALStatus structure type :

> 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.

  - See the LAL Software Documentation for a complete description of these conventions.

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

Rules for LAL functions (2)

- The source code for a simple LAL function such as LALREAL4Divide (in file LALDivide.c) might be:

```
#include <lal/LALDivide.h>
NRCSID( LALDIVIDEC, "$Id: lalspec.tex,v 1.12 2005/05/03 02:37:30 patrick Exp $" );

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 );
}
```

RULES FOR XLAL FUNCTIONS

Rules for XLAL functions (1)

- The goal is to have XLAL functions :
  - Be as flexible as possible in their interface while still requiring strict rules on error reporting.
  - Be intended to be “lightweight” functions that can be used internally within the LAL library.
  - Do not have some of the burdens of LAL functions.
  - Do not have a status structure.
    1. XLAL functions cannot call LAL functions.
    2. XLAL functions can be “lightweight.”
    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)

Rules for XLAL functions (2)

XLAL functions shall not call LAL functions.

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

Four kinds of XLAL 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).

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

OTHER LIBRARIES (REQUIRED FOR LAL)

Other libraries required for LAL

- LAL is not a stand-alone library:
  - Two other libraries are required to build and use LAL
    - "Fastest Fourier Transform in the West (version 3)" FFTW3 library
      - 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.
    - "GNU Scientific Library" GSL library.
      - 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.

Data format

- There are currently two official exchange data formats within the LSC:
  - XML-based “LIGO Lightweight” LIGOlw format
  - binary “Interferometric Gravitational Wave Detector Data Frame Format” or “Frame” format.

  - Libraries with routines that are specialized for I/O with these two formats are also available.

LALSupport, LALMetaIo, LALFrame

- LALSupport library :
  - Basic file I/O routines
- LALMetaIo library :
  - I/O routines that are used to read/write the LIGO lightweight data format. These routines use the METAIO library routines as their engine.
- LALFrame library :
  - I/O routines that are used to read/write the Frame data format. These routines use the FRAME library routines as their engine