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

Document Type: Specification	**LIGO-T990030-07 - E**	02.28. 2000

LIGO Data Analysis System - Numerical Algorithms Library Specification and Style Guide

LIGO Laboratory & LIGO Scientific Collaboration

Distribution of this draft:

LIGO and LSC

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 20B-145
Cambridge, MA 01239
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

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

CHANGE RECORD				
Revision	***Date***	***Authority***	***Pages Affected***	***Item(s) Affected***
		Initial draft	All	All
	Feb. 2000		most	

Organization/Group	***Name***	***Signature***	***Date***
LIGO Laboratory	LDAS Group Leader		
	LDAS Software Task Leader		
	Directorate		
LSC Chairs	ASIS		
	DCSA		
	Detector Characterization		
	Spokesman		

1 TABLE OF CONTENTS

TABLE OF CONTENTS -- continued

LIST OF TABLES

2 INTRODUCTION

The LIGO/LSC Algorithm Library [LAL] for the analysis of data generated by interferometric gravitational wave detectors (IGWD) is a collaborative effort involving the LIGO Laboratory [LL] and contributors from the LIGO Scientific Collaboration [LSC]. This specification has evolved out of the recognition for the need of a standard definition of how the code which will be produced looks and behaves.

It is the intent of LL and LSC to share and encourage the use of these algorithms by other (international) projects wishing to adhere to a common set of software coding standards. It is hoped that by using a standard design for procedural algorithms, consistent analyses of data by different groups of individuals can be promoted more easily.

In order to be more inclusive of future collaborators, the LL Data Group undertook to promote the use of ANSI standard C programming style.

LL and LSC will promote a continued evolution of this standard through formal configuration control, scheduled updates and releases, and code maintenance. An *ad hoc* working group with representatives from LL and LSC has formed to develop this specification and the first coded algorithms. Eventually, it is expected that this will become a formal working group of LSC and it will have formal control over the contents of this specification as it evolves.

2.1. Purpose

This specification formally defines the LIGO/LSC Algorithm Library [LAL].

2.2. Scope

This document specifies the interface between the LDAS software environment and numerical filter libraries (and other code modules) contributed by users for LIGO data analysis. It is also intended to show a user of LDAS how to add new analysis functionality by creating new functions. This is not a comprehensive document explaining how to write functions in general, but rather is a specification for I/O, exception handling and other interfaces needed to allow a piece of code to function correctly within LDAS. Contributed software may only be written in C. This specification anticipates the eventual incorporation of the contributed software into a C++ environment. However, in the absence of a fully defined design specification for future C++ components, developers are requested to limit their present contributions to ANSI standard C.

LAL will be written assuming IEEE/ASCII compliant hardware and software is used to analyze interferometer data.

This standard specifies the organization of LAL components, including many C structures which are used to define I/O behavior.

This specification also defines rules to which new extensions and revisions are required to conform.

2.3. Applicability

The LIGO Laboratory and the LSC will work to ensure that all developed hardware and software systems support LAL. All participating groups will analyze data using LAL-compatible software. The LAL software shall be available in the public domain, subject only to the standards and controls defined herein.

3 OVERVIEW

LDAS is the analysis environment being developed for LL and LSC. It consists of a layered and highly modular architecture employing a steering language or scripting commands, coupled to C++ as the compiled language. The paradigm being employed is that of using MPI based parallel computing to complete the computationally intensive numerical analysis of data. The MPI architecture will involve use of procedural algorithms and functions to manipulate the data. These algorithms, or filters, are expected to be contributed by a larger group of individuals than those responsible for the LDAS overall architecture and implementation. Because of this it is both desirable and necessary to define explicitly a set of interfaces to which a number of individual researcher can write code having a common “look and feel”.

It is the intent to develop and maintain a dynamically loaded LAL. In working to design and build such a library, there are the following advantages:

- C can be used
- one algorithm can be used by all users
- one can utilize capability written by other people very easily into his or her own work

This document focuses on the programming aspects of creating a new procedural algorithm. The document, *Getting Started with LDAS, LIGO-T99XTBD*, is a manual on LDAS, and it contains other references for in depth information. **Table 1** presents a list of relevant design documentation for LDAS.

For now, the supported development environment is the latest version of *egcs* and *GNU’s gcc* and *ccc*. Together, these contain all flavors of C. However, the *GNU gcc* compiler is more robust and stable than the equivalent compiler from *egcs* at the time of this writing. If you prefer to use other environment for the module development, use it until you complete the testing of your code, and do the final build using *GNU gcc*. In general *gcc* is recommended for C and *egcs* is recommended for C++. If your code conforms to the standards described anthill document, the final build will not be a problem.

Table 1 List of Applicable Documents

Description	*Document ID*
LIGO Documentation	
LDAS White Paper	LIGO-M970065
LDAS Design Requirements Document	LIGO-T970159.
LDAS Conceptual Design Document	LIGO-T970160.
LDAS Preliminary Design Document	LIGO-T990001
LDAS System Software Specification for C, C++ and Java	LIGO-T970211
Data Format Specifications	
Specification of a Common Data Frame Format for Interferometric Gravitational Wave Detectors (IGWD)	LIGO-T971030
LIGO Lightweight Data Format Specification	LIGO-T980091.
LIGO Metadata, Event and Reduced Data Requirements	LIGO-T980070.
LDAS Software Specifications[a]:	
LDAS Users Manual	LIGO- T99xTBD
FrameAPI Baseline Requirements	LIGO-T980011.
FrameAPI.tcl source code map -- frameAPI.tcl	on-line TclDoc
FrameAPI.tcl emergency procedures source code map -- frameEmProc.tcl	on-line TclDoc
FrameAPI.tcl operator procedures source code map -- frameOpProcs.tcl	on-line TclDoc
MetadataAPI Baseline Requirements	LIGO-T980119
DataConditioningAPI Baseline Requirements	LIGO-T990002
Non LIGO Documentation	
Enough Rope to Shoot Yourself in the Foot: Rules for C and C++ Programming, Allen I.Holub	McGraw-Hill 1995

a. link accessible via http://docuserv.ligo.caltech.edu/~prince/LDCG_lsc/LDCG.html. Note that some of these documents are still evolving.

4 CONVENTIONS

4.1 Coding style

The design of code presented here reflects the style from Allen I.Holub's book [see **Table 1**]. Header and source code examples are provided in **Appendix A** and **Appendix B**.

Variables must begin with a *lowercase* letter; types must begin with an *Uppercase* letter. Names combining multiple words must have subsequent words *capitalized*: theNewVariable, TheNewType. Macros (#define) must be all *UPPERCASE*. Compound macro names will use underscores if clarity requires: THE_NEXT_MACRO. Function names should start with a capital letter, e.g. MyFunction(). The files (modules) containing the function (and the acompanying .h file) should also begin with a capital letter, e.g. MyModule.c and MyModule.h.

4.2 Physical and numerical constants

There will be an effective LIGO *namespace* for defining the set of physical constants to be used within LDAS. Physical constants will be stored in the header file **LALConstants.h.** This is being distributed with the LAL library releases. All constants are declared according to the following style:

```
#define LAL_CONSTANTNAME_STANDARD value            /* units or description */
```

Examples from **LALConstants.h:**

```
#define LAL_PI         3.14159265358979323846264338327950291L   /* pi  */
#define LAL_RSUN_SI    6.960e08                                  /*solar radius, m*/
#define LAL_SOLMASS_SI 1.9892e30                                /* solar mass, kg */
```

All constants have the reserved prefix LAL_. The constants have a suffix to denote the system of units in which they are defined.

The file **LALConstants.h** is being shipped with the LAL software. A method will be provided to extract the physical constants database as a lightweight data object (XML) to allow this data set to be easily transported.

5 LAL DATA TYPES

Data types may be considered in a hierarchical grouping:

1. Atomic or primitive data types -- language-specific types:
   - int
   - float, etc.
2. Higher level (aggregate) constructs:
   - vectors

- arrays or matrices
- lists, etc.
- sequences or series (time-shares, frequency spectra, etc.)

3. Custom C structures:
   - structures of parameters
   - input or output structures for function I/O, etc.
   - exception handling structures

Items #1 are language [and also platform] dependent. Language specific implementations may be found for items #2 and we will define the LAL versions to correspond as closely as possible to existing constructs. Items #3 are specific to LAL usage. Each of these will be discussed below.

Structures shall be defined according to the following template:

```
typedef struct
tag<Name>
{
   ...;
   ...;
}
<Name>;
```

Where <Name> is replaced by the struct's name. Note the tag name must be used with all structs. (Writing the typedef and the tag-Name in column zero is a GNU convention, and not a LAL requirement; however, much of the code in the library adheres to this convention.)

One variable definition per type declaration is prefered; however a few closely related variables can be declared on the same line. This allows ease of reading and maintenance. It allows each line to have a single comment that pertains to the declaration:

```
   TYPE  variableName;     /* helpful or useful comment */

   INT4  length;           /* number of elements */
   INT4  vectorLength;     /* length of each vector in sequence */
   REAL4 *a,*b,*c;         /* temporary pointer variables  */
   REAL4 *a,b;             /* DO NOT COMBINE POINTERS AND VARIABLES! */
```

5.1 Primitive or "atomic" data types

To permit LAL code to be transported to various hardware platform (e.g., 32, 64 or 128 bit machines), we will adopt the convention described in the LIGO-VIRGO frame specification. To each C/C++ data type there will be assigned a *CAPITALIZED* LAL data type name. These will be defined in **LALDatatypes.h**:

```
...
typedef        char            CHAR
typedef        short           INT2
typedef        unsigned char   BOOLEAN
...
typedef        struct          {
                               REAL4 re;
                               REAL4 im;
                               } COMPLEX8;
typedef        struct          {
```

```
REAL8 re;
REAL8 im;
} COMPLEX16;
```

...

The C data types listed in **Table 2** are recognized. This table is a *subset* (except for the introduction of the type BOOLEAN) of the data type table from **Section 4.2** of **LIGO-T970130.** The COMPLEX4 and COMPLEX8 data types are necessarily structures.

Table 2 LIGO data types for algorithm software

Data Class	C/C++ Data Type	Length (Bytes)	Comments
CHAR	char	1	Character (signed or unsigned is machine dependent)
UCHAR	unsigned char	1	Unsigned character
BOOLEAN	unsigned char	1	Unsigned character
INT2	short or int	2	Signed integer, Range: (-2^{15}, $2^{15}-1$)
UINT2	unsigned short or unsigned int	2	Unsigned integer
INT4	int or long	4	Signed integer, Range: (-2^{31}, $2^{31}-1$)
UINT4	unsigned int or unsigned long	4	Unsigned Integer
INT8	long or longlong	8	Signed integer, Range: (-2^{63}, $2^{63}-1$)
UINT8	unsigned long or unsigned longlong	8	Unsigned integer
REAL4	float	4	IEEE-defined single precision floating point number
REAL8	double	8	IEEE-defined double precision floating point number
Composite Data Types (structures)			
COMPLEX8	Pair of REAL4	8	Complex real number, two single precision floats, stored as a pair: (real, imaginary)
COMPLEX16	Pair of REAL8	16	Complex real number, two double precision floats, stored as a pair: (real, imaginary)

Although the names do not correspond to the names used in FRAME, the underlying philosophy is that there shall be a LAL atomic data type corresponding to every FRAME atomic data type. The important feature of these data types is that they are of specified length, e.g. UINT4 shall be 4 bytes in length, period. This is enforced by the macros in **LALDatatypes.h**.

In the current implementation, the byte-size of the atomic data types is cleverly enforced. At configure-time, a short c-program is run. This program figures out what the byte-lengths are, and writes the information to a file which is included in **LALDatatypes.h**.

5.2 Aggregate constructs of primitive data types

This list is extensible and it is expected that it will be augmented over time. These definitions will also be included in **LALDatatypes.h**. Indexing convention for multi-dimensional arrays will follow the C convention of row-major ordering. Table 3 lists the objects defined below.

Table 3 LIGO data objects [relevant section numbers are shown in table headings]

Data Class	*LIGO Names*	*Comments*	*Comments*
5.1 Atomic -- See Table 2			
5.2 Aggregates			
Vectors	<datatype>Vector	Footnote[a]	Aggregates capture only numerical data useful for computation (e.g.,bytes) no units or physical information is provided at this level
Array	<datatype>Array	Footnote[a]	
Sequences	<datatype>Sequence	Footnote[a]	
	<datatype>VectorSequence	Footnote[a]	
	<datatype>ArraySequence	Footnote[a]	
5.3 Structures			
Time	LIGOTime	A `struct` identifying GPS time.	Physical units or dimensions are encapsulated in the structures.
Series	<datatype>TimeSeries <datatype>FrequencySeries	Examples: time series, spectra, etc.	
	<datatype>TimeVectorSeries <datatype>FrequencyVectorSeries	Example: time series of a vector quantity.	
	<datatype>TimeArraySeries <datatype>FrequencyArraySeries	Example: time series of a matrix quantity.	
	<datatype>TableSeries	Example: time series for a group of objects which are best represented by a table	
Transfer Functions	<datatype>FTransferFunction	List of {f,y,z} triplets for H[f]; {y,z} correspond to {M, φ } or {Re,Im} of H[f]	
	<datatype>ZPGFilter	Pole-zero-gain representation for H[z]	

a. Initially <datatype> will be taken by default to be ONLY from the following list: {CHAR,REAL4, REAL8, COMPLEX8,COMPLEX16, INT2, INT4,INT8,UINT2,UINT4,UINT8}. Additional types may be added when it is shown that they are needed.

5.2.1 Vectors

Vector is a one-dimensional object that corresponds to a collection of `length = N` data elements of the same data type, taken from **Table 2** above.

```
typedef struct
tag<datatype>Vector
{
   UINT4        length;   /* number of elements */
   <datatype> *data;      /* pointer to data of type <datatype> from Table 3,
                             footnote a */
}
<datatype>Vector;
```

Here and elsewhere below <datatype> can be any of the types in **Table 3**, footnote a. **Structs** defined with a <...> prefix will be enumerated in **LALDatatypes.h** for each corresponding data type that is needed. So, for example, the following vector data types will appear: CHARVector, INT2Vector, COMPLEX8Vector, etc. The need for explicit typing follows because C, unlike C++, does not support template data type definitions. Alternative methods using **enum** statements are possible; however, these, unlike the "hard-wired" type casting described above provide extensibility at the cost of case checking (if statements) that need to be embedded in the resultant code.

5.2.2 Arrays

Array is a `dim = ndim (>1)` object that corresponds to a collection of `length = ldim1*ldim2*...*ldimNdim` data elements of the same data type, taken from **Table 3**, footnote a, above.

```
struct <datatype>Array
{
   UINT4Vector *dimLength;    /* vector of <ndim> UINT4 scalars for the
                                 lengths for each of the dimensions */
   <datatype>  *arrayData;    /* pointer to data of type <datatype> from
                                      Table 3, foot note a */
};
```

The discussion at the end of **Section 5.2.1** applies. A vector may also be represented by one dimensional array; however when represented as a vector, the dimension is implicitly assumed to be 1.

5.2.3 Sequences

A sequence (or a series) is a list of `length = N` *compound objects*. The *compound objects* may be either vectors or arrays. Note that a sequence of scalars is represented by the vector object in section **Table 5.2.1** above. All elements of the sequence must have the same identical structure. All data elements are of the same data type, taken from **Table 3**, footnote a, above.

```
typedef <datatype>Sequence <datatype>Vector /* sequence of scalars is the
                                               same as a vector */
typedef  struct
tag<datatype>VectorSequence
{
   UINT4      length;          /* number of elements */
   UINT4      vectorLength;    /* length of each vector in sequence */
```

```
   <datatype>  *data;          /* pointer to data of type <datatype> from
                                  Table 3, footnote a */
}
<datatype>VectorSequence;
```

```
typedef  struct
tag<datatype>ArraySequence
{
   UINT4       length;        /* number of elements */
   UINT4       arrayDim;      /* dimension of each array in sequence */
   UINT4       *dimLength;    /* length of each dimension of array */
   <datatype>  *data;          /* pointer to data of type <datatype> from
                              Table 3, footnote a */
}
<datatype>ArraySequence;
```

The discussion at the end of **Section 5.2.1** applies. A vector sequence may also be represented by one dimensional array sequence; however when represented as a vector sequence, the dimension is implicitly assumed to be 1 for the elements of the sequence.

The indexing for a sequence of *compound objects* will run through the internal indices of the objects before going to the next object in the sequence:

$$\left\{ \begin{bmatrix} h_1[t_1] \\ \cdots \\ h_N[t_1] \end{bmatrix}, \begin{bmatrix} h_1[t_2] \\ \cdots \\ h_N[t_2] \end{bmatrix}, \begin{bmatrix} h_1[t_3] \\ \cdots \\ h_N[t_3] \end{bmatrix}, \ldots, \begin{bmatrix} h_1[t_M] \\ \cdots \\ h_N[t_M] \end{bmatrix} \right\} \Rightarrow$$

$$\{h_1[t_1], \ldots, h_N[t_1], h_1[t_2], \ldots, h_N[t_2], h_1[t_3], \ldots, h_N[t_3], h_1[t_M], \ldots, h_N[t_M]\}$$

5.3 LIGO structured data types

This list is extensible and it is expected that it will be augmented over time. These definitions will be included in **LALDatatypes.h**.

5.3.1 Time

5.3.1.1 Time stamps

GPS time will be supported within data structures. Other time standards may be captured as comments. To indicate this, the time structure will have “GPS” in its name. There will be a set of time inter-conversion utilities that can be used to go between various standards.

```
typedef  struct
tagLIGOTimeGPS
{
   INT4 gpsSeconds;
   INT4 gpsNanoSeconds;
}
LIGOTimeGPS;
```

Note the capitalization convention for acronyms: When the convention calls for an acronym to start with lower case, the entire acronym is written in lower case (e.g.. `INT4 gpsSeconds`).

When the convention calls for the acronym to start with an upper case letter, the entire acronym is capitalized (e.g. `tagLIGOTimeGPS`). We have tried to consistently apply this throughout LAL.

5.3.1.2 Multiple time stamps

M*ultiple* time stamps (e.g., for a *vector* of strains, each coming from an instrument in a different geographical location) can be accommodated as a C array of type **LIGOTimeGPS**:

```
LIGOTimeGPS gpsTimeList[10];            /* a list of 10 LIGOTime structures */
```

5.3.2 Sequences in time

5.3.2.1 TimeSeries

The structure **TimeSeries** is used to represent a sequence of samples taken at uniformly spaced intervals of time. A **TimeSeries** object has the following attributes:

- name of series
- time of epoch - time at which the *earliest* sample in the series was acquired;
- number of samples in series
- deltaT offset between samples (reciprocal of sample rate)
- ***time offset units will be in seconds***
- units of values recorded in samples
- time sequence of data

```
typedef  struct
tag<datatype>TimeSeries
{
   CHAR               *name;            /* user assigned name */
   LIGOTimeGPS        epoch;           /* epoch of first series sample */
   REAL8              deltaT;          /* sample spacing in time */
   REAL8              f0;              /* base frequency,!=0 if
                                          heterodyned series  */
   CHARVector         *sampleUnits;    /* units for sampled quantity */
   <datatype>Vector   *data;           /* the data */
}
<datatype>TimeSeries;
```

5.3.2.2 TimeVectorSeries

The structure **TimeVectorSeries** is used to represent a sequence of vectors taken at uniformly spaced intervals of time. A **TimeVectorSeries** object has the following attributes:

- name of series
- time of epoch - time at which the *earliest* sample in the series was acquired;
- number of samples in series
- deltaT offset between samples (reciprocal of sample rate)
- ***time offset units will be in seconds***
- units of values recorded in samples
- length of each vector in series (hidden in the ‘‘Vector’’ structure)
- time vector sequence of the data

```
typedef  struct
tag<datatype>TimeVectorSeries
{
   CHAR                        *name;         /* user assigned name */
   LIGOTimeGPS                 epoch;         /* times of first elements in
                                                 vector series */
   REAL8                       deltaT;        /* sample spacing in time -- same
                                                 for all elements */
   REAL8                       f0;            /* base frequency,!=0 if
                                                 heterodyned series */
   CHARVector                  *sampleUnits;  /* units o sampled quantities */
   <datatype>VectorSequence    *data;         /* the data */
}
<datatype>TimeVectorSeries;
```

5.3.2.3 TimeArraySeries

The structure **TimeArraySeries** is used to represent a sequence of vectors taken at uniformly spaced intervals of time. A **TimeArraySeries** object has the following attributes:

- name of series
- time of epoch - time at which the *earliest* sample in the series was acquired;
- number of samples in series
- deltaT offset between samples (reciprocal of sample rate)
- ***time offset units will be in seconds***
- units of values recorded in samples
- length of each vector in series (hidden in the array structure)
- dimensions of array (hidden in the Array structure)
- lengths of each dimension in array (hidden in the Array structure)
- time array sequence of data

```
typedef  struct
tag<datatype>timeArraySeries
{
   CHAR             *name;         /* user assigned name */
   ligoTimeGPS      epoch;        /* times of first elements in array series */
   REAL8            deltaT;         /* sample spacing in time - same for all
                                       elements */
   REAL8            f0;           /* base frequency,!=0 if heterodyned series*/
   CHARVector       *sampleUnits; /* units for sampled quantities */
   <datatype>ArraySequence  *data; /* the data */
}
<datatype>TimeArraySeries;
```

The discussion at the end of **Section 5.2.1** applies with regard to typecasting **<datatype>Time<seriestype>Series** [generic name for all three types] for each of the LIGO data types. As a minimum, the following **Time*Series** types are needed initially:

- **INT2Time<seriestype>Series** (for 16 bit ADC data)
- **REAL4Time<seriestype>Series**
- **REAL8Time<seriestype>Series**

5.3.3 Sequences in frequency

5.3.3.1 FrequencySeries

The structure **FrequencySeries** is used to represent result of a Fourier transformation on a **TimeSeries** object. It may have both negative and positive frequency components, depending on the value of the starting frequency parameter. A **FrequencySeries** object has the following attributes:

- name of series
- time of epoch - time at which the *earliest* sample in the [pre-transformed] data was acquired;
- number of samples in series, N (Hidden in the vector structure)
- deltaF offset between samples
- ***frequency units will be in Hertz***
- first frequency in series.
- The series spans the interval {f0,f0+deltaF,....,f0+(N-1)*deltaF}
- units of values recorded in samples
- frequency vector sequence of data

```
typedef  struct
tag<datatype>frequencySeries
{
   CHAR               *name;          /* user assigned name */
   ligoTimeGPS        epoch;          /* time value of first array element */
   REAL8              f0;             /* first frequency in sample */
   REAL8              deltaF;         /* sample spacing in time */
   CHAR               *sampleUnits;   /* units for sampled quantity */
   <datatype>Vector   *data;          /* the data */
}
<datatype>frequencySeries;
```

FrequencySeries can contain any of the following types of spectra:

- two-sided frequency series, real or complex (according to vector data type declaration)
- one-sided frequency series
- power-spectrum (one-sided real frequency series)

5.3.3.2 FrequencyVectorSeries

The structure **frequencyVectorSeries** is used to represent result of a Fourier transformation on a **timeVectorSeries** object. It may have both negative and positive frequency components, depending on the value of the starting frequency parameter. A **frequencyVectorSeries** object has the following attributes:

- name of series
- time of epoch - time at which the *earliest* sample in the [pre-transformed] data was acquired;
- number of samples in series, N (Hidden in the Vector structure)
- deltaF offset between samples
- ***frequency units will be in Hertz***
- first frequency in series.

The series spans the interval {f0,f0+deltaF,....,f0+(N-1)*deltaF}

- units of values recorded in samples
- length of each vector in series (Hidden in the Vector Structure)
- frequency vector series of data

```
typedef  struct
tag<datatype>frequencyVectorSeries
{
   CHAR                    *name;           /* user assigned name */
   LIGOTimeGPS             epoch;           /* time values of first vector
                                               element */
   REAL8                   f0;              /* first frequency in sample */
   REAL8                   deltaF;          /* sample spacing in time */
   CHARVector               *sampleUnits    /* units for sampled quantities */
   <datatype>VectorSequence  *data;         /* the data */
}
<datatype>frequencyVectorSeries;
```

5.3.3.3 FrequencyArraySeries

The structure **FrequencyArraySeries** is used to represent result of a Fourier transformation on a **TimeArraySeries** object. It may have both negative and positive frequency components, depending on the value of the starting frequency parameter. A **FrequencyArraySeries** object has the following attributes:

- name of series
- time of epoch - time at which the *earliest* sample in the [pre-transformed] data was acquired;
- number of samples in series, N
- deltaF offset between samples
- ***frequency units will be in Hertz***
- first frequency in series.

The series spans the interval {f0,f0+deltaF,....,f0+(N-1)*deltaF}

- units of values recorded in samples
- dimensions of array (Hidden in Array structure)
- lengths of each dimension in array (Hidden in Array structure)
- frequency array series of data

```
typedef  struct
tag<datatype>FrequencyArraySeries
{
   CHAR                    *name;           /* user assigned name */
   ligoTimeGPS             *t0;             /* time values of first vector
                                               element */
   REAL8                   f0;              /* first frequency in sample */
   REAL8                   deltaF;          /* sample spacing in time */
   CHARVector              *sampleUnits;    /* units for sampled quantities */
   <datatype>ArraySequence  *data;          /* the data */
}
<datatype>FrequencyArraySeries;
```

The discussion at the end of **Section 5.2.1** applies with regard to typecasting **<datatype>Frequency<seriestype>Series** [generic name for all three types] for each of the

LIGO data types. As a minimum, the following **<datatype>Frequency<seriestype>Series** types are needed initially:

- **REAL4Frequency<seriestype>Series;**
- **REAL4Frequency<seriestype>Series;**
- **COMPLEX8Frequency<seriestype>Series;**
- **COMPLEX16Frequency<seriestype>Series**.

5.3.4 Series of n-tuples

The structure **TableSeries** is used to represent ordered n-tuple data for which, for example, sampling rate is not a fixed value. **TableSeries** would be used to represent calibration data taken at logarithmically spaced frequency intervals. A **TableSeries** object has the following attributes:

- name of series
- time of epoch - time at which the original data which were transformed were acquired;
- number of samples in object, N (Hidden in Vector structure)
- number of elements per sample - length of each element (Hidden in Vector structure)
- units of values recorded in samples
- vector sequence table of data

```
typedef  struct
tag<datatype>TableSeries
{
   CHAR*                        *name;          /* user assigned name */
   LIGOTimeGPS                  epoch;          /* time value of first
                                                      array element */
   CHARVector                   *sampleUnits;  /* vector with units for
                                                sampled quantities */
   <datatype>VectorSequence     *data;         /*  the n-tuple data */
}
<datatype>TableSeries;
```

The discussion at the end of **Section 5.2.1** applies with regard to typecasting **TableSeries** data types for each of the LIGO data types. As a minimum, the following **TableSeries** types are needed initially:

- **REAL4TableSeries**;
- **REAL8TableSeries**;
- **COMPLEX8TableSeriese**
- **COMPLEX16TableSeries**.

5.3.5 Transfer functions

5.3.5.1 Frequency domain

The structure **FTransferFunction** is used to represent H[s]:

- name of transform
- list of frequencies
- list of magnitude, phase, *or*
- list of real, imaginary

```
enum {XferMag, XferXY} XferType;                /* R*exp[i*phi] vs. x+iy
                                                   representation for Xfer */

typedef  struct
tag<datatype>FTransferFunction
{
   XferType                XferRepresentation;  /* Bode representation for
                                                   real-imaginary */
   CHAR                    *name;    /* user assigned name */
   CHARVector              *HNames; /* e.g., "f_Hz, H_mag, H_phi_radian\n" */
   <datatype>VectorSeries  *hData;  /* the H[s] as 3-tuples */
}
<datatype>FTransferFunction;
```

The discussion at the end of **Section 5.2.1** applies with regard to typecasting **FTransferFunction** for each of the LIGO data types. As a minimum, the following **FTransferFunction** types are needed initially:

- **REAL4FTransferFunction**;
- **REAL8FTransferFunction**.

5.3.5.2 Zeros, poles and gain representation

The structure **ZPGFilter** is used to represent a transfer functions as a list of zeroes, poles, and a gain. This is a factored version of **ZTransferFunction**.

- name of transform
- gain, G (real)
- poles, $p_{k\ (complex)}$
- zeroes, $z_{k\ (complex)}$

```
typedef  struct
tag<datatype>ZPGFilter
{
   CHAR                    *name;         /* user assigned name */
   <datatype>Vector        *zeros;        /* the zeros */
   <datatype>Vector        *poles;        /* poles */
    <datatype>              gain;          /* filter gain */
}
<datatype>ZPGFilter;
```

The discussion at the end of **Section 5.2.1** applies with regard to typecasting **ZPGFilter** for each of the LIGO data types. As a minimum, the following **ZPGFilter** types are needed initially

- **COMPLEX8ZPGFilter**;

- **COMPLEX16ZPGFilter**.

6 FILTER ALGORITHM DESIGN

6.1 Procedural function style and usage

The following are guidelines for writing analysis functions for LIGO data. The general style should be consistent with the style specification LIGO-T970211. Function definitions are designed to follow the DSL/Globus Coding Standard from ANL (the MPICH group, see http://www-fp.mcs.anl.gov/dsl/). In cases where what is described below differs from T970211, the present document takes precedence.

Functions written according to these guidelines will be simpler to verify, to maintain and to incorporate into general analysis systems. The prototypical analysis function is referred to in these guidelines as **LIGOFunction()**.

1. All functions should be specified using the following format:

```
[static] <return type>
<function name> (
                <param type 1> <param name 1>,
                <param type 2> <param name 2>,
                ...
                <param type N> <param name N>
                )
{
  <code block(s)>
} /* <function name>() */
```

2. **LIGOFunction()** is of **type void** and shall contains four and only **four** arguments (thus spaketh the software committee):

```
void LIGOFunction(
                Status *stat,
                LIGOFunctionOutStruct *output,
                LIGOFunctionInStruct *input,
                LIGOFunctionParamStruct *params
                );
```

The first argument is a pointer to a **status structure** (described below, identical for all functions). The second and third arguments are pointers to an **output structure** and a **input structure** respectively. The **input** and **output structures** will specified for classes of functions having common behaviors. The intent is to provide *defined and controlled* structures for all LAL functions to permit to the greatest extent possible ease-of-use and simple data passing between functions which are called sequentially.

The fourth argument is a **parameter structure** which can be used to pass other types of data, including re-entrant behavior information, to the function. Code developers are encouraged to use LIGO standard data types (described above) where possible within these structures.

Explanation: This makes it easier to extend or to add extra functionality to procedures. When additional arguments are needed they can be added as members of the input, output or parameter structures without modifying any existing code that calls **LIGOFunction()**.

3. **LIGOFunction()** shall return control to the scope from which it was called.

The **status structure** is used to report the completion status of the function when it returns. Its format is:

```
typedef struct
tagStatus
{
   INT4                    statusCode;
   const CHAR              *statusDescription;
   volatile const CHAR     *Id;
   const CHAR              *file;
   INT4                    line;
   struct tagStatus        *statusPtr;
   INT4                    *level;
}
Status;
```

If **LIGOFunction()** completes successfully, **statusCode** should be set to zero. Upon abnormal termination of **LIGOFunction**, **statusCode** must be assigned a non-zero value. Values for **statuscode** must be documented and assigned symbolic names in **LIGOFunction.h**. **statusDescription** is a pointer to a static character string defined in **LIGOFunction()** that provides a brief summary of the problem. **Id** is a static character string defined in **LIGOFunction()** that contains rcsid version (rcsid) information. The method should be provided as part of the debugging process (see #5 below) to provide this information to **stdout** or some other designated output. The field **file** contains the name of the module where the error occurred. **line** contains the line number in module where the error occurred.

The status structure definition is recursive to permit status to be returned from various levels of nested function calls (i.e., functions called within functions, which are called within functions,...). **level** keeps track of how many levels deep the problem actually occurred.

A few negative values for the statusCode have been reserved for some generic failures:

Table 4 Generic abnormal termination codes (statusCode)

statusPtr value	***description***
0	normal termination
-1	recursive error: function failed due to error in deeper subroutine
-2	status pointer passed to function had a nonnull statusPtr field
-4	function unable to allocate `statusPtr`
-8	`statusPtr` could not be dealloacted

Explanation: If functions always return, the flow of control is always controllable at the highest level. The status code and description allows the top level to identify and resolve possible problems. Version, author and date information are easily available by ensuring that the static character string with this information is part of **Status.**

4. Direct calls to **malloc()**, **free()**, **calloc()** and **realloc()** are not allowed.

The custom handlers will take several additional arguments, including a short text description of the use of the memory being allocated/freed.

They are replaced by functions LALMalloc(), LALFree(), LALCalloc(), LALRealloc(). (See file `LALMalloc.h`.)

Explanation: This simplifies tracking memory usage and memory leak identification.

5. **LIGOFunction()** should free all memory that it allocates, except for storage for variable length output parameters.

Explanation: This avoids memory leaks. Persistent intermediate storage and fixed length output parameters should be allocated by the calling function.

6. Functions and procedures must refer to:

```
extern INT4 debuglevel;
```

when deciding whether to print debugging information. Legal values for **debuglevel** are 0,1, or 2 (or greater). If it is 0, then no debugging information will be printed. If 1, then some debugging information should be printed. If 2 (or greater), then verbose information should be provided. Analysis functions may not modify **debuglevel**.

Explanation: allows calling program to provide diagnostic info if needed to understand unusual behavior.

Warning: do not test the value of **debuglevel** within critical floating point loops. The presence of an integer compare/branch instruction often interferes with efficient floating-point execution.

7. Function should be in a file **LIGOFunction.c** and come with a header file **LIGOFunction.h**. *Small* sets of *related* functions may be grouped together into a single (**File.c**, **File.h)** pair.

The header file should define function prototypes and structures and have comments describing all inputs and outputs of the function (including any file I/O) as well as the processing performed by the function. The header file should protect against multiple includes, and have a C++-compatible structure.

Explanation: this will make it easier to exchange useful functions.

Comment: in the future we may ask that comments be structured in a way that supports automatic documentation generation (doc++, for example).

8. File input/output using **fopen()**, **fclose()**, **fprintf()**, etc. is not allowed.

Custom file I/O functions will be provided. A function should close all files that it opens, except for files that are explicitly passed to the calling function by a FILE pointer in the output structure.

Explanation: file access may not be available (permissions, space) or appropriate on given machines. The custom file I/O routines will deal with this.

9. Each function must come with a stand-alone program **LIGOFunctionTest.c** which can be linked to **LIGOFunction()** and tests it.

LIGOFunctionTest returns **SUCCESS** if the function works and **FAILURE** if it does not. In the event of an error, **LIGOFunctioTest** should print out the expected and actual result values of the particular test it performs. It should also print the contents of **statusCode**, **statusDescription** and **Id** which are returned by **LIGOFunction()**.

Explanation: this is a simple way to provide validation. Any necessary data should be either contained in **LIGOFunctionTest.c** or computed by it. **SUCCESS** and **FAILURE** are system-dependent return codes defined in the analysis procedure header file.

10. Allocation of significant amounts of memory, should use the **custom LALmalloc()** rather than automatic stack variables.

Explanation: many machines and shells do not support large stacks. Typical stack sizes are 8 to 64 Mbytes. It is easy to blow the stack and this can be hard to identify with debuggers and other tools.

11. Debugging/information/warning messages should be printed with a **custom replacement** for **printf()** and **fprintf(stderr,...)**.

This function will be provided and will take the same arguments as **printf()** and possibly other arguments.

Explanation: this allows debugging/information/warning messages to be handled in different ways, depending on the operating environment and conditions. For example, they might be logged, sent immediately to the user, ignored, etc.

12. Developers should endeavor to use LIGO standard data structures whenever possible. These will be described above.

Explanation: General use of standard data structures will allow the development of a suite of reusable manipulation functions and ease the interfacing between functions.

13. **LIGOFunction()** should be re-entrant.

In other words, it should not contain variables that save internal state information between function invocations. If such state variables are needed, then they must be included in one of the argument structures.

Explanation: Functions that are not re-entrant *cannot* be invoked by different routines without special precautions. They are also more difficult to maintain.

14. Aliasing (i.e., allowing two structures point to or share the same memory address) is expressly prohibited. An exception to this is the case where (mutually exclusive) memory sharing is effectively supported by ANSI C (e.g., **unions**).

Explanation: It becomes difficult to keep track of whether memory is being pointed to and, consequently, difficult to avoid memory leaks or “amnesia (freeing memory being used). Code maintenance becomes more difficult when aliasing is permitted.

6.2 LAL specific data structures

6.2.1 Status

The status structure design was described in #2 above. Once again it is:

```
typedef struct
tagStatus
{
   INT4                 statusCode;
   const CHAR           *statusDescription;
   volatile const CHAR  *Id;
   const CHAR           *file;
   INT4                 line;
   struct tagStatus     *statusPtr;
   INT4                 level;
}
Status;
```

6.2.1.1 statuscode

statusCode = 0 for successful completion; **statusCode** != 0 otherwise. A table of symbolic values must be provided in the function header file. Examples are:

```
...
#define OK                      0       /* successful execution */
#define DIVIDE_BY_ZERO          1       /* flag for dividing by zero */
#define OUT_OF_RANGE_POSITIVE   2       /* value is unexpectedly large */
...
```

6.2.1.2 statusDescription

statusDescription is a pointer to a static character string defined in **LIGOFunction()**. It provides a brief summary of the problem. Examples are:

```
...
constant CHAR *statusDescription;
...
statusDescription = “”;                                 /* statuscode = OK */
```

```
statusDescription = “attempted divide by zero in LIGOFunction”;
                                                /* statuscode = DIVIDE_BY_ZERO */
...
statusDescription = “value too large in LIGOFunction”;
                                              /* statuscode = OUT_OF_RANGE_POSITIVE */
```

6.2.1.3 rcsid[]

rcsid[] is a static character string defined at checkout by the version control system and embedded in **LIGOFunction().** It contains the full path name of the RCS file, revision number, date, author, state identifier [release, alpha, etc.] and locker (if locked). Locker contains the loginID of the user (if any) who had locked the code for the purpose of making revisions at the time the present version was exported. This construct corresponds to the CVS standard and shall be used for all **LIGOFunction**s. For example, checking into CVS a code fragment containing the following line:

```
...
static const CHAR rcsid[] = “$Header$”;
```

would be converted by CVS to the following upon exporting or checking out the same code fragment:

```
static const CHAR rcsid[] = “$Header: /ldas/api/genericAPI/so/LIGOFunction.c,v
1.5.1 1999/11/28 23:20:28 Beta kent$”
```

6.2.2 LIGOFunctionIOStruct (Not available as of Feb. 2000)

LIGOFunctionIOStruct is designed to handle input and output of data for **LIGOFunction().** The detailed definition of **LIGOFunctionIOStruct** depends on **LIGOFunction()**. It will be documented along with the documentation of the specific **LIGOFunction()**. The input and output structure designs will use a common data structure which is specific to the function or class of similar functions. An example might be:

```
typedef  struct tagLIGOFunctionInStruct{
   <datatype>Vector*       inputData;
   ...
                           }LIGOFunctionInStruct;
typedef  struct tagLIGOFunctionOutStruct{
   <datatype>timeSeries*   outputData;
   ...
                           }LIGOFunctionOutStruct;
```

6.2.3 LIGOFunctionParamStruct (Not available as of Feb. 2000)

LIGOFunctionParamStruct is designed to accommodate the parameter data for **LIGOFunction().** The detailed definition of **LIGOFunctionParamStruct** depends on **LIGOFunction()**. It will be documented along with the documentation of the specific **LIGOFunction()**. An example might be:

```
typedef  struct tagParamStruct{

   INT2                    callsToLIGOFunctionCounter;
                 /* counter used to determine possible re-entrant behavior */
   REAL8                   filterSeed;                                    ;
                           /* seed for re-entrant random number generator */
   REAL8                   filterCoefficients[10];
                                        /* array of 10 filter coefficients */
   <datatype>Vector*       data;
                           /* vector of LIGO data types needed for filter */
   ...
                  }ParamStruct;
```

7 LAL DEVELOPMENT TOOLS

To develop code for LAL requires the use of the following software development, documentation and testing tools:

7.1 Development tools:

- GNU CVS: version 1.10 or greater.
- GNU egcs: version 1.2.0 or greater.
- GNU make: version 3.72 or greater.
- GNU m4: version 1.4 or greater.

7.2 Documentation tools:

- PERCEPS
- PDF (generated by any means).

7.3 Testing tools:

- Each LAL code element will have its own main() test program provided by the contributor.

8 DIRECTORY ORGANIZATION

All files associated with a specific LAL software module will reside in a single directory, whose organization is described below.

8.1 Root directory

Every LAL software component will have a named, designated root directory. That directory contains:

- all files necessary to configure the build of the task (i.e., autoconfig files and Makefiles)
- automake file, Makefile.am to recursively make subdirectories.
- A directory called package/**packages** that contains the subdirectories that in turn contain

the documentation, header, source and test files for that component. The subdirectories (packages) will be named as follows and in turn contain:

- **/doc**: all the documentation associated with the component.
- **/include**: all the header files associated with this component. Header files must conform to the format and style described in this document.
- **/src**: all the source files associated with the component. Source files must conform to the format and style described in this document.
- **/test**: test scripts and all supporting files (but not documentation) associated with component-level tests. Component level tests must conform to the format and style described in this document.

In addition to these directories, there will be CVS subdirectories at all levels. Files in the CVS directories will be configured and maintained by CVS. The only files permitted in these directories are those created by CVS and the only modifications permitted on files in these directories can made through CVS.

There may be additional subdirectories needed for some software components, whose purpose and function will be assigned by a Software Coordinator.

8.2 LAL Component Documentation

Each completed LAL module must be accompanied by documentation. Documentation will be written in *LaTeX* format. The LaTeX output can be converted to PDF format to support on-line viewing (through, e.g., a web-browser) and off-line (through a printed manual) documentation formats.

All documentation will have a uniform format:

1. **Purpose**: an overview describing the component's purpose or function.
2. **Algorithms**: describes the algorithms used to provide the desired functionality.
3. **Arguments**: describes the input and output arguments and data formats for the component. For each input/output argument, the domain/range should be described.
4. **Operating instructions**: describes how this component is called. Example code fragments should be provided, showing the construction of the input arguments, the call, error checking, and de-construction of the output arguments.
5. **Options**: describes all options which affect the input, output or function.
6. **Accuracy**: describes the guaranteed (expected?) accuracy of the results.
7. **Error conditions**: describes the returned error codes, what triggers them, and the argument state when an error exit is taken. All error conditions which are tested for should be described here.
8. **Tests**: describes the test suite that accompanies the component. All tests which are performed should be documented individually. Required tests are described in Section 8.5.
9. **Uses**: cross-references to other LAL routines that are used directly by this component.
10. **References**: Bibliographic references and cross-references to other documentation.

8.3 Header Files

Each component source file has a corresponding header file. The purpose of a header file is to encapsulate logically related information required for the use of the corresponding sub-component. Information not required by other routines using this component should not appear in the header.

Header files will conform to the format in **Appendix A** and contain the following information, in the order presented (Comment field with file name, author, revision, etc., as specified above);

1. Include-loop protection.
2. Includes. This header may include other headers; if so, they go immediately after include-loop protection. Includes should appear in the following order:
   - Standard library includes;
   - LDAS includes;
   - LAL includes;

   Includes should be double-guarded (see **Appendix B***).*
   Header file version string (from CVS; see **Appendix B***).*

3. Macros. But, note that macros are deprecated.
4. Extern Constant Declarations. These should not be present unless a specific waiver has been granted.
5. Extern Global Variables. These should also not be present unless a specific waiver has been granted.
6. Structure, enum, union, etc., typedef.
7. Functions Declarations (i.e., prototypes).

Note: no executable code appears in a header file.

8.4 Source Files

Each component source file will have a corresponding header file. The purpose of a header file is to encapsulate logically related information required for the use of the corresponding sub-component. Header files will conform to the format in **Appendix A** and contain the following information, in the order presented:

1. Prolog: an extended comment field containing summary information about the source code module (see **Appendix B** for format and contents); Source file version string (from CVS).
2. Include directives. These should be guarded and appear in the following order:
   - Standard library includes;
   - LDAS includes;
   - LAL includes.
3. Each source file must include at least its own header. Includes should be guarded (see **Appendix B**).
4. Constants and enumerated types used only internally;
5. Type declarations (i.e., typedefs) used only internally;
6. Function macros for which a waiver has been granted.

7. Extern global variable declarations for which a waiver has been granted;
8. Static global variables for which a waiver has been granted;
9. Static function declarations for which a waiver has been granted;
10. Function definition(s).

8.5 Component level tests

Each completed task is accompanied by a suite of verification tests. A verification test is meant to evaluate a component to determine if it satisfies the requirements imposed on it. Every component will have several requirements it must satisfy, which will be specified when the component is assigned to a developer. The developer is responsible for providing tests that demonstrate these requirements are met by the delivered component.

As a general rule, a test suite should involve tests from at least three categories:

- Mainline tests, which demonstrate that the routine correctly acts on commonly encountered input data;
- Inside-edge tests, which demonstrate that the routine correctly acts on input data that are barely legitimate;
- Outside-edge tests, which demonstrate that the routine correctly acts on input data that are barely illegitimate.

Note that, in the case of illegitimate data, correct action involves raising and returning the appropriate error conditions.

The test suite provided with each component is meant to test just the functioning of that component. The test suite delivered with each component should not attempt to test or diagnose the functioning of other components.

In addition to tests that verify that the component meets the requirements specified at the time the component is assigned, the developer is encouraged to provide other tests that verify more detailed aspects of the component's behavior.

Each LAL code element will have its own main() test program provided by the contributor. Included must be a script (in a unix shell language) which executes main() under various conditions with data sets provided along with the code element. Results will be returned in a formatted report [format definition to be provided later]. Tests must execute and exit CLEANLY.

9 CONFIGURATION CONTROL

LAL software will be delivered with makefiles which, as a minimum, enable installation, compilation and execution of code elements within the following environments:

- linux [Redhat 6.0 or later] on Intel hardware;
- Solaris 7 on SUN hardware.

Each subdirectory in a distribution that contains something to be compiled or installed should come with a file **Makefile.am** file, from which automake will create **Makefile.in**, from which configure will create a Makefile in that directory.

Makefiles should conform to the GNU standards and conventions, which are found in the GNU document **standards.info**.

9.1 Version control

Version control of all LAL files will be coordinated by the LSC Software Coordinator. When a LAL module is assigned for development, its directory tree will be created in the CVS repository. The assigned developer(s) will be provided read/write access to that part of the repository. Additional access to other parts of the repository will be provided as needed.

The LSC software coordinator will assign a major version number to the developed product, and may assign minor numbers and branches as well. Releases will be managed by CVS tags.

See also Section 11.

10 LIGO SOFTWARE ARCHIVES

10.1 LDAS directory structure (to be replaced)

Table 5 presents the LDAS distribution tree structure as of this writing. This structure will continue to evolve.

Table 5: LDAS distribution directory structure

root	include	ilwd
		framecpp
		general
		genericAPI
		ospace
		⋮
	lib	genericAPI
		frameAPI
		metadataAPI
		⋮
	bin	
	macros	frame
		metadata
		metascripts
		⋮

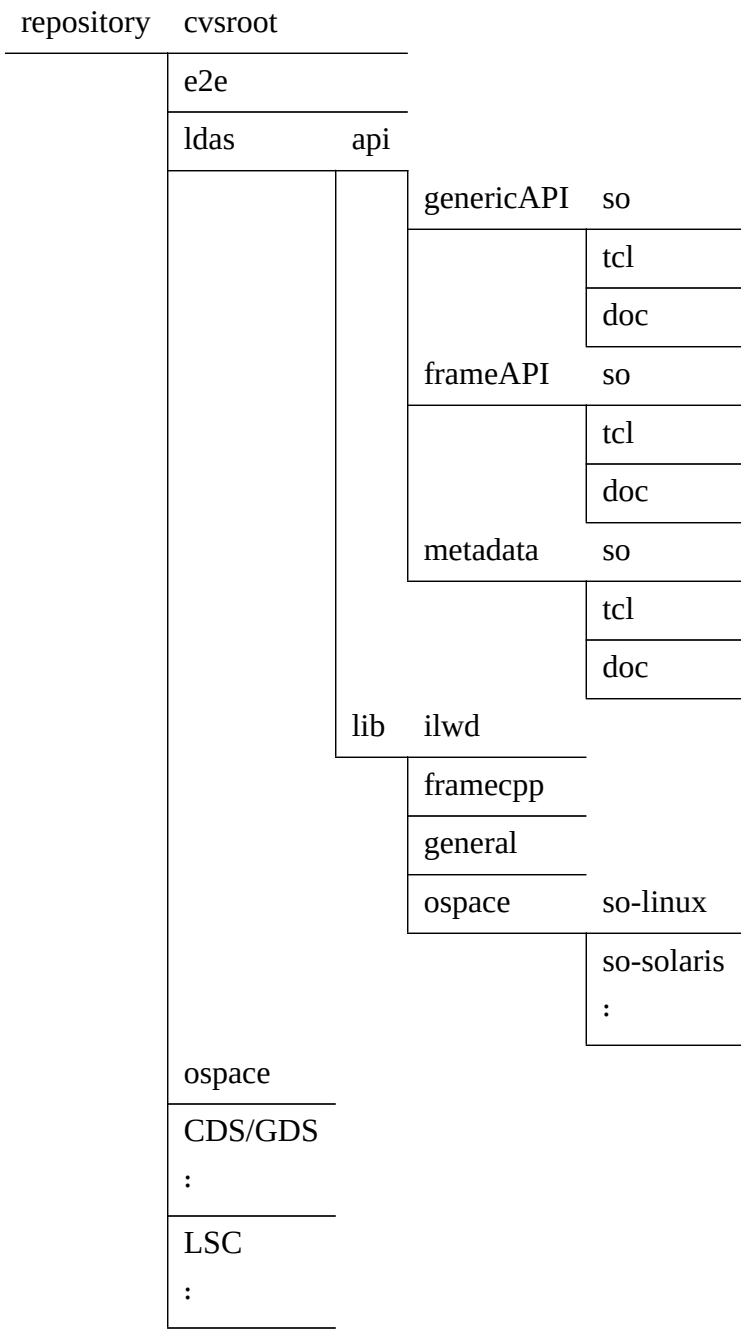
10.2 CDS/GDS directory structure

To be provided by CDS

10.3 CVS repository structure (to be replaced)

Table 6 presents the CVS repository tree structure as of this writing. This structure will continue to evolve.

Table 6: CVS repository directory structure



11 RULES FOR REVISION

LL and LSC will jointly maintain both the specification for LAL software and also all associated software libraries.

The numbering scheme for future releases of LAL shall be a two digit number with the two digits separated by a decimal point (.): e.g., LAL Release “X.Y”.

Individual software components in the library shall also be identified by version number. The version specification for the software libraries shall also be in the form “X.Y”.

X = version number. This is incremented whenever major changes are introduced. If X is incremented, Y is reset to 0.

Y = revision number. This is incremented whenever one or more of the following changes are made: (i) software error fixes; (ii) enhancements in existing functionality; (iii) modification or addition of structures not addressed by X above.

11.1. Requests for changes

LL and LSC will maintain a web page (address **To Be Announced)** for submitting requests for changes and for providing for releases of code.

11.2. Change control

LAL software will be placed under joint configuration control by LL and LSC using UNIX/CVS.

Updates will be provided by the following basis.

a. Change requests will be reviewed jointly by LL and LSC on a regular basis.

b. Those changes which are selected for incorporation shall be assigned for implementation to respective groups.

c. All changes will be validated and verified using a prescribed test procedure.

d. Once available, the new release will be distributed via the LL and LSC web site. All affected documentation will be revised to show changes.

e. A history of revisions shall be maintained and made available to users.

APPENDIX A EXAMPLE/TEMPLATE HEADER FILE

```
/*-----------------------------------------------------------------------
 *
 * File Name: example.h
 *
 * Author: A. Hacker
 *
 * Revision: $Id$
 *
 *-----------------------------------------------------------------------
 *
 * NAME
 * example.h
 *
 * SYNOPSIS
 * #include "example.h"
 *
 * DESCRIPTION
 * Example header file prolog.
 *
 * DIAGNOSTICS
 */

/*
 * Header contents go here, in order specified:
 *
 * 1. Prolog (Comment field with file name, author, revision, etc., as
 *    specified above)
 * 2. include-loop protection (see below). Note the naming convention!
 */

#ifndef _EXAMPLE_H
#define _EXAMPLE_H

/*
 * 3. Includes. This header may include others; if so, they go immediately
 *    after include-loop protection. Includes should appear in the following
 *    order:
 *    a. Standard library includes
 *    b. LDAS includes
 *    c. LAL includes
 *    Includes should be double-guarded!
 */

#ifndef _STDLIB_H
#include <stdlib.h>
#ifndef _STDLIB_H
#define _STDLIB_H
#endif
#endif

#ifndef _LALCONSTANTS_H
#include "LALCONSTANTS.h"
#ifndef _LALCONSTANTS_H
#define _LALCONSTANTS_H
#endif
#endif

/*
 * 4. Header file version string (from CVS; see below). Note the string name.
 */
```

```
static char *EXAMPLEH = "$Id$";

/*
 * 5. Macros. But, note that macros are deprecated.
 *
 * 6. Extern Constant Declarations. These should not be present unless a
 *    specific waiver has been granted.
 *
 * 7. Extern Global Variables. These should also not be present unless a
 *    specific waiver has been granted.
 *
 * 8. Structure, enum, union, etc., typedefs.
 *
 * 9. Functions Declarations (i.e., prototypes).
 */

#endif
```

APPENDIX B EXAMPLE/TEMPLATE SOURCE FILE

```
/*-----------------------------------------------------------------------
 *
 * File Name: example.c
 *
 * Author: J. Random Hacker
 *
 * Revision: $Id$
 *
 *-----------------------------------------------------------------------
 *
 * NAME
 * example
 *
 * SYNOPSIS
 * (void) example()
 *
 * DESCRIPTION
 * Example source file prolog.
 *
 * DIAGNOSTICS
 * (Abnormal termination conditions, error and warning codes summarized
 * here. More complete descriptions are found in documentation.)
 *
 * CALLS
 * (list of LAL, LDAS, other non-system functions/procedures called.
 *
 * NOTES
 * (Other notes)
 *
 *-----------------------------------------------------------------------
 */

/* 1. Prolog: an extended comment field containing summary information
 *    about the source code module (see above);
 * 2. Source file version string (from CVS). Note the string name.
 */
static char *EXAMPLEC = "$Id$";
/*
 * 3. Include directives. These should be guarded and appear in the
 *    following order:
 *    a. Standard library includes;
 *    b. LDAS includes;
 *    c. LAL includes.
 *    Each source file must include at least its own header. Includes should
 *    be guarded, as in
 */

#ifndef _STDLIB_H_
#include <stdlib.h>
#define _STDLIB_H_
#endif

#ifndef _MATH_H_
#include <math.h>
#define _MATH_H_
#endif

#ifndef LDASDCAPI_H
#include "LDASDCAPI.h"
#define LDASDCAPI_H
#endif
```

```
#ifndef EXAMPLE_H
#include "example.h"
#define EXAMPLE_H
#endif

/*
 * 4. Constants, enumerated types, structures, etc., used only internally;
 * 5. Type declarations ({\em i.e.,} {\tt typedefs\/}) used only
 *    internally;
 * 6. Function macros for which a waiver has been granted;
 * 7. Extern global variable declarations for which a waiver has been
 *    granted;
 * 8. Static global variables for which a waiver has been granted;
 * 9. Static function declarations for which a waiver has been granted;
 * 9. Function definition(s).
 */
```

APPENDIX C EXAMPLE/TEMPLATE MAKEFILES

See code distribution for examples.