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

Document Type: Specification	**LIGO-T990030-08 - E**	07.18. 2000

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

Bruce Allen, Kent Blackburn, Jolien Creighton, Sam Finn,
Albert Lazzarini and Alan Wiseman

Distribution of this draft:

LIGO and LSC

This is an internal working note of the
LIGO Laboratory and the
LIGO Scientific Collaboration.

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	
	July 2000		All	

Organization	***Name***	***Signature***	***Date***
LIGO Laboratory	Directorate		
LSC	Software Coordinator		
	spokesperson		

1 TABLE OF CONTENTS

TABLE OF CONTENTS -- continued

LIST OF TABLES

2 INTRODUCTION

2.1 The goal of the LAL software specification

The LIGO Laboratory [LL] and the LIGO Scientific Collaboration [LSC] are developing the LIGO/LSC Algorithm Library [LAL] for analyzing data from interferometric gravitational-wave detectors. The LL and LSC wish to share this software with other projects and invite other (international) groups to contribute to this library. **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 details in this specification flow naturally from this goal.

1. More programmers know C than C++; therefore, in order to maximize the number of users and contributors, the LL Data Group decided to use ANSI standard C for the LAL. Similarly, we don't want contributors to have to climb multiple learning curves just to master the tools necessary to write LAL code; therefore we specify a minimal list of development tools in **Section 9**.
2. The output of one programmer's routine is likely to be the input of someone else's routine. To make this exchange easy, we specify some reusable data structures for input and output. (We also **require** developers to use them whenever possible.) These are given in **Section 4**.
3. One programmer must be able to use, understand, test and debug another programmer's code; therefore we establish some coding conventions (**Section 3**), a uniform layout for the source code (**Sections 5 and 6**), and the specifications for the documentation (**Section 7**). In particular, we define the namespace conventions in **Section 3**, and we explain the details of reporting errors in **Section 5**.
4. It is essential that users and developers know the pedigree of the routines they are using; therefore we have defined a version control system (CVS) for the library in **Section 8**.
5. Using a standard design for the software will (hopefully) make it easier to test the routines and compare data analysis results of different groups.
6. This code project will grow and evolve. This makes it impossible to foresee all the necessary code requirements. Therefore, the LL and LSC will continue to jointly update and maintain this specification. The rules for this procedure are given in **Section 8.**
7. In order to facilitate collaboration, the LSC Software Coordinator will ensure that the code is publicly (and easily) available to users and developers.

2.2. The scope of the LAL specification

This document formally defines the LIGO/LSC Algorithm Library [LAL]. This is not a comprehensive document explaining how to write LAL functions, rather it lays out general rules for code writing. Eventually, we may write a C++ specification for LAL; however, until we have such a specification, code must be written in ANSI standard C.

2.3. Applicability of LAL software

The LIGO Laboratory and the LSC will work to ensure that all developed hardware and software systems support LAL. In turn, <u>all participating groups will be required to analyze LIGO data using LAL-compliant software.</u> The LAL software shall be available in the public domain, subject only to rules in this document. LAL will be written assuming IEEE/ASCII compliant hardware and software is used to analyze interferometer data.

2.4 How does the LAL fit into the LDAS?

The 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 (e.g. Tcl). The scripting language will execute compiled C++ code which will use MPI based parallel computing to do the numerically intensive data analysis. [See http://www.ldas.ligo.caltech.edu and Table 1 for detail information on LDAS.]

The current plan is to use procedural algorithms and functions (i.e., LAL routines written in C) wrapped in C++ code to manipulate the data. These functions will be imported into the C++ code as a dynamically loaded (shared object) library. These dynamically loaded LAL functions will actually perform the data analysis.

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	LIGO-T971030
LIGO Lightweight Data Format Specification	LIGO-T980091.
LIGO Metadata, Event and Reduced Data Requirements	LIGO-T980070.
LDAS Software Specifications[a]:	
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

Table 1 List of Applicable Documents

Description	*Document ID*
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. Links accessible via httt://www.ldas.ligo.caltech.edu and http://www.ligo.caltech.edu/LIGO_web/dcc/docs. Note that some of these documents are still evolving.

3 LAL CODING STYLE

3.1 LAL namespace conventions

3.1.1 The rationale behind the namespace rules

1. The naming convention should make it easier for someone (besides the author) to understand the code.
2. The naming convention should help avoid internal (intra-LAL) name conflicts.
3. LAL will be used in conjunction with other libraries; therefore the naming convention should help avoid conflicts with non-LAL software packages and system routines.

3.1.2 The namespace rules

1. **Names combining multiple words** must have subsequent words *capitalized*: `theNewVariable`, `LALTheNewType`. The names tend to be long enough as it is; therefore we do not use the underscore between words in a name. [Macros are an exception to this rule. See below.]
2. **Variable names** must begin with a *lowercase* letter, e.g. `myVariable`.
3. **Function names** must begin the prefix `LAL`. The remainder of the name should also start with a capital letter, e.g. `LALMyFunction()`. The `LAL` prefix will help keep the LAL namespace from conflicting with other library namespaces. As LAL grows, there is also a risk of stepping on our own namespace; therefore don't use nondescript function names, such as "`LALCorrolate()`" or "`LALFilter()`". Use more specific names, e.g. attach the package name or the header-file name: `LALInspiralFilter()`. [Note: Requiring the `LAL` prefix is a significant change from earlier versions (7 and earlier) of this document. This will require substantial modifications of existing code, but it is necessary.]
4. **Custom data structures** (i.e. structures not specified in this document) must be given names that try to avoid namespace conflicts. The name should start with an *Uppercase* letter, e.g. `LALREAL8MyDataType`. We suggest using the prefix `LAL` to avoid collision with other libraries; however this is not a requirement. Another way to avoid conflicts with other packages is to build the name around the Atomic datatype, e.g. `REAL8MyPackageVector`. The discussion

about non-descript function names applies here as well. Also, names without the LAL prefix, can step on system names; therefore don't use words like `time`, `date`, `window`, etc.

5. **Source-code file names** (modules, headers and test programs) should also begin with a capital letter, e.g. `MyModule.c` and `MyHeader.h`.
6. **Acronyms in the name**: 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 should never see `gPS` or `Gps`.
7. **Macros** (`#define`) must be all *UPPERCASE*. Compound macro names will use underscores if clarity requires: `THE_NEXT_MACRO`. [This is only exception to the no-underscore rule.]
8. **Error codes** (`statusCode` and `statusDescription`) have a special name convention. See **Section 4.5.**
9. **Package names** should be all *lowercase*.

3.2 Physical and numerical constants

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         3.1415926535897932384626433832795029L /* 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. If there are constants that should be there, but are not, contact the LSC Software Coordinator.

3.3 Style for type declarations

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! */
```

4 LAL DATA TYPES

In order to facilitate sharing of data between LAL routines and passing data form LAL to non-LAL library functions (e.g. the rest of LDAS) we define a number of generic data structures. You are required to use these structures whenever possible in your code. We recognize that we can't plan for every contingency, so, if you find that there are structures that are not included, but would have widespread use if they were available, please tell the LSC Software Coordinator.

4.1 Defining data types

Structures shall be defined according to the following template:

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

Where `<Name>` is replaced by the struct's name. The tag is optional. (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.)

4.2 "Atomic" data types

To permit LAL code to be transported to various hardware platforms (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 `LALAtomicDatatypes.h`. See **Table 2**.[The structures `COMPLEX8` and `COMPLEX16` are also included in our list of atomic data types.]

```
typedef struct
{
        REAL4 re;
        REAL4 im;
}
COMPLEX8;

typedef struct
{
        REAL8 re;
        REAL8 im;
}
COMPLEX16;
```

Table 2 LAL 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: $(-2^{31}, 2^{31}-1)$

Table 2 LAL data types for algorithm software

Data Class	C/C++ Data Type	Length (Bytes)	Comments
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)

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 `LALAtomicDatatypes.h`.

4.3 Aggregate constructs of atomic data types

This list is of aggregate constructs of atomic data types may be augmented in the future. These definitions will 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 LAL data objects [relevant section numbers are shown in table headings]

Data Class	*LAL Names*	*Comments*	*Comments*
4.2 Atomic -- See Table 2			
4.3 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]	
4.4 Structures			
Time	LIGOTime	A `struct` identifying GPS time.	Physical units or dimensions are encapsulated in the structures.

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

Data Class	*LAL Names*	*Comments*	*Comments*
Series	<datatype>TimeSeries <datatype>FrequencySeries	Examples: time series, spectra, etc.	
	<datatype>SequenceOfTimeSeries <datatype>SequenceOfFrequencySeries	Example: two polarizations of a gravitational wave signal.	
	<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,UCHAR,REAL4, REAL8, COMPLEX8,COMPLEX16, INT2,INT4,INT8,UINT2,UINT4,UINT8`}. Additional types may be added when shown to be needed.

4.3.1 Vectors

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

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

}
<datatype>Vector;
```

Here and elsewhere <datatype> can be any of the types in **Table 3**, footnote a. Structures defined with a <...> prefix will be enumerated in **LALDatatypes.h** for each corresponding data type that is needed. 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.

4.3.2 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>  *data;         /* pointer to data of type <datatype> from
                               Table 3, foot note a                   */
};
```

The discussion at the end of **Section 4.3.1** applies.

4.3.3 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 4.3.1** 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  struct
tag<datatype>VectorSequence
{
   UINT4      length;          /* number of vectors in the sequence */
   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 arrays in sequence        */
   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 4.3.1** applies.

4.3.3.1 The packing order for a VectorSequence or ArraySequence

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. For example, here is a sequence of `data->length=M` vectors, each of `data->vectorLength=N`.

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

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

In **Section 4.4.2.2.1** we define a structure where the packing is in the other order.

4.4 LAL structured data types

This list of time structures will be augmented as the need arises. The definitions are in **LAL-Datatypes.h**.

4.4.1 Time

4.4.1.1 Time stamps

There is a specific data structure to store GPS time. To indicate this, the time structure will have "GPS" (or gps) in its name.

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

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 LIGOTimeGPS structures */
```

4.4.2 Sequences in time

4.4.2.1 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
- epoch - time at which the *earliest* sample in the series was acquired
- deltaT - offset between samples (reciprocal of sample rate). ***(Time offset units will be seconds.)***
- units of values recorded in samples
- the data is stored in a `<datatype>Vector` structure. This structure contains:
  - the number of elements in the sequence `data->length`
  - the data itself is in `data->data[]`

```
typedef  struct
```

```
tag<datatype>TimeSeries
{
   CHAR               name[LALNameLength]; /* 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;
```

4.4.2.1.1 The `name[]` field in structured data types

The name field will be an array at most `LALNameLength` characters long. `LALNameLength` will be set in `LALDatatypes.h`. Currently, the value is set to `64`, although we could make change it. In previous version (7 and earlier) of this document `name` was a `CHAR*`, and it didn't specify what form the name should take. This made it cumbersome to write generic routines that freed the memory. [This method of handling the name is the same as much of the rest of LDAS.]

4.4.2.2 SequenceOfTimeSeries

The structure **SequenceOfTimeSeries** is used to represent a sequence of time series, each of which starts at the same time, e.g. the two time-series representing the two polarizations of gravitational wave. A **SequenceOfTimeSeries** object has the following attributes:

- name of series. See **Section 4.4.2.1.1**.
- time of epoch - time at which the *earliest* sample in the series was acquired
- deltaT offset between samples (reciprocal of sample rate).(time offset units will be seconds). ***(Time offset units will be seconds.)***
- units of values recorded in samples
- the data is stored in a `<datatype>VectorSequence` structure. This structure contains:
    - the length of the sequence (i.e. the number of series) is in `data->length`
    - the number of elements in each time series `data->vectorLength`
    - the data it self in `data->data[]`

Note: The structure **SequenceOfTimeSeries** is similar to the **TimeVectorSeries** structure. The distinction is in the order of the packing in `*data`. **See Sections 4.4.2.2.1**.

```
typedef  struct
tag<datatype>SequenceOfTimeSeries
{
   CHAR                      name[LALNameLength];/* 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>VectorSequence *data;              /* the data                    */
}
<datatype>SequenceOfTimeSeries
```

4.4.2.2.1 The Packing order for a SequenceOfTimeSeries

As an example of how the packing goes, consider two time series s0[t] and s1[t]:

```
data->length       = 2       ;      /* number of series                  */
```

```
data->vectorLength = 1024     ;      /* number of elements in each series */

data->data[0]      = s0[0]    ;
data->data[1]      = s0[1]    ;
  ...
data->data[1023]   = s0[1023] ;


data->data[1024]   = s1[0]    ;
data->data[1025]   = s1[1]    ;
  ...
data->data[2047]   = s1[1023] ;
```

4.4.2.3 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. See **Section 4.4.2.1.1**.
- epoch - time at which the *earliest* sample in the series was acquired;
- deltaT offset between samples (reciprocal of sample rate). (Time offset units will be seconds.)
- units of values recorded in samples
- the data is stored in a `<datatype>VectorSequence` structure. This structure contains:
  - The number of times when data is taken is in `data->length.` This the total number of vectors. All the elements of each vector are evaluated at the same time in this structure.
  - The number of elements measure at each time is in `data->vectorLength`
  - The actual data values are stored in `data->data[]`.

Note: The packing of **TimeVectorSeries** is described in **Section 4.3.3.1** [Compare **Section 4.4.2.2.1.]**

```
typedef  struct
tag<datatype>TimeVectorSeries
{
   CHAR                      name[LALNameLength];    /* 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;
```

4.4.2.4 TimeArraySeries

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

- name of series. See **Section 4.4.2.1.1**.
- epoch - time at which the *earliest* sample in the series was acquired;
- deltaT - offset between samples (reciprocal of sample rate). ***(Time offset units will be***

seconds.)

- units of values recorded in samples
- the data is stored in a `<datatype>ArraySequence` structure. This structure contains:
  - The number of time samples is stored in `data->length` This is the number of arrays.
  - The dimension of each array is stored in `data->arrayDim`
  - The length of each dimension of the array in `data->dimLength`
    [Note: all the values of each array are taken at same time.]
  - The data is stored in `data->data[]`

```
typedef  struct
tag<datatype>timeArraySeries
{
   CHAR           name[LALNameLength]; /* 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 4.3.1** applies with regard to typecasting **<datatype>Time<seriestype>Series** [generic name for all three types] for each of the LAL 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**

4.4.3 Sequences in frequency

4.4.3.1 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. See **Section 4.4.2.1.1**.
- epoch - time at which the *earliest* sample in the [pre-transformed] data was acquired;
- 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
- the data is stored in a `<datatype>Vector` structure. This structure contains:
  - The number elements in the series is stored in `data->length`
  - The data itself is in `data->data[]`

```
typedef  struct
tag<datatype>frequencySeries
{
   CHAR                 name[LALNameLength]; /* 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)

4.4.3.2 SequenceOfFrequencySeries

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

- name of series. See **Section 4.4.2.1.1**.
- time of epoch - time at which the *earliest* sample in the [pre-transformed] data was acquired;
- first frequency in series.
- deltaF offset between samples. ***<u>(Frequency units will be in Hertz.)</u>***
  The series spans the interval {f0,f0+deltaF,....,f0+(N-1)*deltaF}
- units of values recorded in samples
- the data is stored in a `<datatype>VectorSequence` structure. This structure contains:
  - the length of the sequence (i.e. the number of series) is in `data->length`
  - the number of elements in each time series `data->vectorLength`

Note: The structure **SequenceOfFrequencySeries** is similar to the **FrequencyVectorSeries** structure. The distinction is in the order of the packing in `*data`. See **Section 4.4.2.2.1.**

```
typedef  struct
tag<datatype>SequenceOfFrequencySeries
{
   CHAR                        name[LALNameLength];/* 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>SequenceOfFrequencySeries;
```

4.4.3.3 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. See **Section 4.4.2.1.1**.
- epoch - time at which the *earliest* sample in the [pre-transformed] data was acquired;
- first frequency in series.
- deltaF - offset between samples. ***(Frequency units will be in Hertz.)***
  The series spans the interval {f0,f0+deltaF,....,f0+(N-1)*deltaF}
- units of values recorded in samples
- the data is stored in a `<datatype>VectorSequence` structure. This structure contains:
    - The number of elements measured at each frequency is in `data->vectorLength`.
    - The number of frequencies where data is taken is in `data->length.`
    - The actual data values are stored in `data->data[]`

Note: The structure **SequenceOfFrequencySeries** is similar to the **FrequencyVectorSeries** structure. The distinction is in the order of the packing in `*data`. See **Section 4.4.2.2.1.**

```
typedef  struct
tag<datatype>frequencyVectorSeries
{
   CHAR                       name[LALNameLength]; /* 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;
```

4.4.3.4 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. See **Section 4.4.2.1.1**.
- epoch - time at which the *earliest* sample in the [pre-transformed] data was acquired;
- first frequency in series.
- deltaF offset between samples. ***(Frequency units will be in Hertz.)***
  The series spans the interval {f0,f0+deltaF,....,f0+(N-1)*deltaF}
- units of values recorded in samples
- the data is stored in a `<datatype>ArraySequence` structure. This structure contains:
    - The number of frequency samples is stored in `data->length`
    - The dimension of each array is stored in `data->arrayDim`
    - The length of each dimension of the array in `data->dimLength`
      [Note all the values in each array are evaluated at a single frequency.]
    - The data is stored in `data->data[]`

```
typedef  struct
tag<datatype>FrequencyArraySeries
{
   CHAR                        name[LALNameLength];/* 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 4.3.1** applies with regard to typecasting **<datatype>Frequency<seriestype>Series** [generic name for all three types] for each of the LAL 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**.

4.4.4 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. See **Section 4.4.2.1.1**.
- 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[LALNameLength]; /* 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 4.3.1** applies with regard to typecasting **TableSeries** data types for each of the LAL data types. As a minimum, the following **TableSeries** types are needed initially:

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

4.4.5 Transfer functions

4.4.5.1 Frequency domain

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

- name of transform. See **Section 4.4.2.1.1**.
- 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[LALNameLength]; /* 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 4.3.1** applies with regard to typecasting **FTransferFunction** for each of the LAL data types. As a minimum, the following **FTransferFunction** types are needed initially:

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

4.4.5.2 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. See **Section 4.4.2.1.1**.
- gain, G (complex)
- poles, p_k (complex)
- zeroes, z_k (complex)

```
typedef  struct
tag<datatype>ZPGFilter
{
   CHAR                     name[LALNameLength];/* 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 4.3.1** applies with regard to typecasting **ZPGFilter** for each of the LAL data types. As a minimum, the following **ZPGFilter** types are needed initially

- **COMPLEX8ZPGFilter**;
- **COMPLEX16ZPGFilter**.

4.5 LALStatus

`LALStatus` is the structure passed to a function to report success or failure.

```
typedef struct
tagLALStatus
{
   INT4                  statusCode;
   const CHAR            *statusDescription;
   volatile const CHAR   *Id;
   const CHAR            *function;
   const CHAR            *file;
   INT4                  line;
   struct tagLALStatus   *statusPtr;
   INT4                  level;
}
LALStatus;
```

4.5.1 The LAL statusCode and *statusDescription fields

The symbolic values must be provided in the header file, and they must be auto-extracted to appear in the documentation. (**statusCode** = 0 for successful termination.)

```
/* the values and names of statusCode for header file MyHeader.h           */
#define MYHEADERH_EOK             0       /* successful execution          */
#define MYHEADERH_EDIVZERO        1       /* flag for dividing by zero     */
#define MYHEADERH_EHIGHRANGE      2       /* value is unexpectedly large   */

/* values and names of messages (statusDescription) for header MyHeader.h  */

#define MYHEADERH_MSGEOK        "Function terminated normally"
#define MYHEADERH_MSGEDIVZERO   "Attempt to divide by zero"
#define MYHEADERH_MSGEHIGHRANGE"value to large in function"
```

Furthermore, using the naming convention illustrated here is required. The **statusCode**'s (error codes) must begin with the header file name (converted to upper case with a trailing `H`) and are appended with `_E<name of error>` (e.g. `MYHEADERH_EDIVZERO`). The corresponding **statusDescripiton**'s (error messages) are the same except they are appended with `_MSGE<name of error>` (e.g. `MYHEADERH_MSGEDIVZERO`). The text string in the message should be a brief description of what went wrong.

4.5.2 The LAL CVS Id string

In each source code file (.h and .c) the version control ''Id'' string will appear twice. [See, e.g., the example header file in Appendix A.] In the author-version block at the top of the file, the string `$Id: MyFile.c$` will be converted by the CVS to something like:

```
$Id: MyFile.c, v 1.7 2000/05/27 05:30:00 jolien $
```

Also in each file we assign the Id string to string constant. This is done in all files with the macro `NRCSID()`. When you first write MyFile.c, you must make the assignment

```
NRCSID(MYFILEC, “$Id: Myfile.c$”);   /* of course MYFILEH for MyFile.h */
```

The CVS will convert this to something like

```
NRCSID(MYFILEC, “$Id: Myfile.c$, v 1.7 2000/05/27 05:30:00 jolien”);
```

The CVS Id string contains the file name, 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. [The only difference between requiring the CVS "Id" string and the CVS "Header" string is that the Header string also gives the absolute path to the file.]

5 LAL FUNCTIONS

5.1 The burning question

Do all the routines that I write really have to obey all the rules for LAL functions? Answer: **If your function is visible in the library, it must obey all the rules given below.** However, many of the requirements below pertain to the interface between LAL functions and the outside world. Inside a given module you may use static functions (i.e. functions that are not visible in the library) that don't jump through all these hoops. Allowing this flexibility is not only friendly, it is computationally sound. Many of the LAL function requirements are time consuming, e.g. allocating the status structure every time you call a LAL function. If we required this to be done every time a simple arithmetic function is called in a loop, the code would take forever to execute. We could require that the arithmetic code be written in-line, and avoid the function call. However this discourages programmers from writing modular code that is easy to maintain.

Don't abuse this flexibility. This is not a license to write code that doesn't conform to the specification, and then dress it up in a wrapper that presents the correct appearance. The Software Coordinator is watching!

5.2 The rules for LAL functions

The following are guidelines for writing analysis functions for LIGO data. The general style should be consistent with the style specification LIGO-T970211. 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, maintain and incorporate into general analysis systems. In the following guidelines, the prototypical analysis function is referred to as `LALFunction()`.

1. `LALFunction()` is of **type void** and shall have a maximum of **four** arguments:

```
void
LALFunction( LALStatus              *status,
```

```
            LALFunctionOutStruct   *output,
            LALFunctionInStruct    *input,
            LALFunctionParamStruct *params
);
```

The first argument is a pointer to a **status structure** (See **Section 4.5.**). This argument is required for all LAL functions, period. The remaining three arguments are optional.

The second and third arguments are pointers to an **output structure** and an **input structure** respectively. Use the LAL datatypes whenever possible for these structures!

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 required to use LAL data types (described above) where possible within the parameter structure.

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 `LALFunction()`.

Admonition: There is a certain amount of ambiguity about what is an input, an output, or a parameter for a function. When you modify a function, don't cheat and try to slip something into parameter block that is clearly an input or output. [The software coordinator is watching!]

2. `LALFunction()` shall return control to the routine that called it.The **status structure** [**Section 4.5**] is used to report the completion status of the function when it returns.

If `LALFunction()` completes successfully, `statusCode` should be set to zero. Upon abnormal termination of `LALFunction()`, `statusCode` must be assigned a non-zero value. Values for `statusCode` must be documented and assigned symbolic names in `LALFunction.h`. `statusDescription` is a pointer to a static character string also defined in `LALFunction.h`**.** This string should provide a brief summary of the problem. A specific syntax and naming convention for the `statusCode` and the `statusDescription` is given in **Section 4.5**. `*Id` is a static character string assigned in `LALFunction()` and defined in `LALFunction.h` that contains CVS information. The field `function` contains the name of the function where the error occurred. `line` contains the line number in module where the error occurred. The field `file` contains the name of the module where the error occurred.

The status structure definition is recursive to permit the 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.

Table 4 shows the negative values for the `statusCode` that have been reserved for some generic failures:

Explanation: If functions always return, the programe flow is controllable at the highest level. The status code and description allows the top level to identify and resolve possible problems.

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

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

Table 4 Generic (ab)normal 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: These simplify tracking memory usage and memory leak identification.

4. `LALFunction()` 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.

5. Functions and procedures must refer to: `extern INT4 lalDebugLevel`;

when deciding whether to print debugging information. The `lalDebugLevel` feature has been considerably enhanced from previous versions of this document. It allows very discriminating choices in what information will be printed. For example, `lalDebugLevel =0` means no information will be printed. `lalDebugLevel = 1` will print only print serious error information, `lalDebugLevel = 3` will print errors and warnings, `lalDebugLevel = 16` will print only memory allocation debugging information. See the code in the LAL release for the full functionality of this feature.

Explanation: allows calling program to make discriminating choices about diagnostic information to understand unusual behavior. Allowing the programmer to select the debugging information to printed is essential: if everything is printed, you can't find what you are looking for.

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

6. The function should be in a `.c` file and come with a `.h` header file.
   Small sets of *related* functions may be grouped together into a single (`File.c`, `File.h`) pair.
   See **Sections 6.2.1 and 6.2.2** for the content and layout of the header and source files.

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

7. 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.

8. Each function must come with a stand-alone test program, which can be linked to `LALFunction()`. See **Section 6.2.3**.

9. 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.

10. 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.

11. Developers should use LAL standard data structures whenever possible. See **Section 4**.

Explanation: It is easier to pass information between functions.

12. `LALFunction()` 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.

13. Aliasing (i.e., allowing two structures to 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.

14. `LALFunction()` should not raise or trap signals.

6 LAL CODE ORGANIZATION

This chapter explains the layout of the code within the LAL. First we give the large-scale structure: the directory tree. Then we describe the finer structure: the required format and content of the individual source files.

In **Chapter 7**, you will notice that the code and the documentation are inextricably entwined: the hierarchy of the code elements (packages, headers, modules) determines the hierarchy of the documentation (chapters, sections, subsections). Even at finer resolution this holds: the contents of the individual source files also matches the content of the individual documentation pieces.

6.1 The big picture: the LAL directory tree

All LAL components (i.e. code, header files, Makefiles, configure scripts, documentation etc.), will reside in a single directory (called `lal/` in this discussion) and its subdirectories. The LSC Software Coordinator and Software Librarian will maintain an official "master copy" of the LAL source in the CVS repository. Loosely speaking, a "release" of the LAL consists of a tar-ball of the master copy of this directory. User can download and install a release on their own machines.

Within this top level directory, their will be a subdirectory (`lal/packages/`) where the analysis code will reside. Within this subdirectory, every LAL software component will have a named directory that contain all files associated with the package (e.g. `lal/packages/inspiral`). The development of "packages" will be the primary way collaborators will contribute to the LAL.

A package subdirectory (e.g. `lal/packages/mypackage`) should have the source files, documentation and Makefiles in the following subdirectories:

- `lal/packages/mypackage/include`: all the header files associated with this component. Header files must conform to the format and style described in S**ection 6.2.1**.
- `lal/packages/mypackage/src`: all the source files associated with the component. They must conform to the format and style described in **Section 6.2.2**.
- `lal/packages/mypackage/test`: test scripts and all supporting files associated with component-level tests. The tests must conform to the format and style described in **Section 6.2.3**.
- `lal/packages/mypackage/doc:` There will be a LaTeX file in this directory capable of assembling a "stand-alone" documentation for this package. There will also be LaTeX file that forms a chapter in comprehensive manual for the entire LAL. Before auto-extraction with **laldoc**, much of the text source for the documentation may reside in the code files. See **Section 7**.

6.1.1 Making LAL code modular

In order to make LAL code easy to use, it should be modular; therefore, as a general rule, packages should have (at most) a few headers in the `/include` directory, (at most) a few related modules should include each header file, and only a few -- closely related -- functions should be in each module.

6.2 The finer picture: the format of LAL code

6.2.1 Header Files

Header files will conform to the format in **Appendix A** and contain the following information, in the order presented.

1. An author and `Id` block. Note, the CVS will supply the file name and version number in the `Id` string. This information must be auto-extracted for inclusion in the documentation.
2. Brief (one sentence) description of the functionality of the header file.
3. A comment block with a Synopsis and description of the functionality supported by this header.
4. Protection against double inclusion.
5. Includes. This header may include other headers; if so, they go immediately after the double-include protection. Includes should appear in the following order:
    - Standard library includes;
    - LDAS includes;
    - LAL includes;
6. Assignment of `Id` string using `NRCSID()`. **See Section 4.5**.
7. Error codes and messages. These must be auto extracted for inclusion in the documentation.
8. Macros. But, note use of macros is discouraged.
9. Extern Constant Declarations. These are strongly discouraged.
10. Structures, enums, unions, typedefs, etc.
11. Extern Global Variables. These are strongly discouraged. Inform the Software Coordinator.
12. Functions Declarations (i.e., prototypes).

Note: no executable code appears in a header file.

6.2.2 Source Files

Source files will conform to the style presented in **Appendix B** and contain the following information in the order presented.

1. An author and `Id` block. Note, the CVS will supply the file name and version number in the `Id` string. This information must be auto-extracted for inclusion in the documentation.
2. Extended comment block that forms the **nucleus of the documentation for this module**. (See **Section 7** for the specific outline.) If the text gets too long and the “code gets lost in the documentation”, you must move the text elsewhere.
3. Includes. These should be guarded and appear in the following order:
    - Standard library includes;
    - LDAS includes;
    - LAL includes.
4. Assignment of `Id` string using `NRCSID()`. See **Section 4.5** for instructions.
5. The code. [The following order is prefered, but there may be exceptional circumstances.]
    (a) Constants, structures (used only internally in this module)
    (b) Type declarations (used only internally)

(c) Macros (discouraged, used only internally)
(d) Extern global variable declarations (Strongly discouraged)
(e) Global variables (Strongly discouraged)
(f) Static function declarations.
(g) The functions that make up the guts of this module. (Remember, to auto-extract the prototypes for inclusion in the documentation.)

6.2.3 Component level tests

Along with each header file there should be an executable that tests every function prototyped in the header file. These executables should extensively (if not exhaustively) test the error condition that can be thrown by a function. The program should report success or failure for all the tests and exit cleanly.

As these executables will not form part of the dynamically loaded library of functions, there is a bit more flexibility in how they are written. For example unix shell scripts that run an executable multiple times with different command line options are allowed. Also keep in mind, these executables should serve as example code on how to use the functions.

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.

In the case of illegitimate data, ‘‘success’’ of the test involves correctly reporting the failure and returning the appropriate error conditions.

7 LAL CODE DOCUMENTATION

Along with any code submission to the LAL library, developers will need to supply documentation. Keep in mind, the documentation, like the code, is a *deliverable*, and the software coordinator will carefully review the documentation to ensure that it adheres to these specifications.

Why don’t we use the LDAS documentation template for LAL code? Most of the LDAS software is written in C++, and therefore the documentation is naturally built around “classes’’. LAL code is written in C, thus the LDAS model doesn’t apply. None the less, our system does mimic the LDAS model as closely as possible by building the LAL documentation around header files and the modules and functions that include them.

7.1 The requirements driving the documentation design

The defining goals of the LAL specification (widespread-use and collaborative-development of the code) lead to a clear requirement for the documentation: **The documentation should not only help the author maintain his or her code, but it should be clear enough that any devel-**

oper can read it and figure out how the code works. If you find yourself saying, ‘‘The easiest way for *me* to maintain *my* code is ...’’, you have missed the point.

The fact that others will need to find their way through the documentation leads naturally to a sensible requirement: **The documentation must have a uniform presentation**. This might be cumbersome in the case of simple functions and restrictive in other cases, but it is still necessary.

The documentation must be accurate. Therefore we have a custom-built documentation tool (`laldoc`) that allows authors to extract code fragments, comments and extended LaTeX source from the code files and import them directly into the documentation.

7.2 LAL documentation rules

The following rules follow naturally from the requirements above:

1. **Documentation will be written in LaTeX.** Reason: (1) The equation-writing capability of LaTeX. (2) It is easy to translate LaTeX to pdf, so the document can be read on the web. (3) Most of the LAL programmers know LaTeX, thus they won't need to learn another typesetting language.
2. **The author and CVS Id block in the code must be auto-extracted from the code and automatically included in the documentation.** Reason: Obvious. It should be clear what version of the code the documentation pertains to.
3. **Error codes and error descriptions must be auto-extracted from the header files and automatically included in the documentation.** Reason: Obvious. There should be no doubt the error information in the documentation is exactly what is in the code. There is a simple tool within `laldoc` for doing the extraction.
4. **Function prototypes must be auto-extracted and included in the documentation.**
5. **All functions must be entered into the LaTeX document index with an `/index{}` command.** Reason: If somebody runs across a function in the code, they should be able to find the documentation by looking it up in the index. The `LAL` prefix on function names should be omitted when putting them in the index.
6. **All non-LAL data structures must be entered into the LaTeX document index with an `/index{}` command.** Reason: Same as functions.
7. **Do not let the code get lost in the documentation.** Using `laldoc` allows one to put the source of the documentation in the source code files; however the text of the documentation can easily grow to be longer than the code itself. If the comment block containing the documentation starts to swamp the code, move some documentation, e.g. put the documentation at the end of the file and use the LaTeX command `/input{}` to build the document.

7.3 The organization of LAL documentation

The organization of the documentation follows the organization of the code. The hierarchy of the code elements determines the hierarchy of the documentation elements. The documentation for each package will form a ***chapter***. The documentation for each header file within the package will form a ***section*** of the package chapter. The documentation for each module that includes that header file will form a ***subsection*** of the header section. Similarly, the test modules associated with each header file will also form a ***subsection*** of the header section. The documentation of the individual code pieces also closely follows from the code architecture. This design makes it easy to build the documentation with `laldoc`. The References will come at the end of each package chapter. [This method of organizing documentation around headers and functions is similar to the way books on C organize the documentation of the standard libraries.]

The fact that packages form chapters also means that they independently form reasonably self-contained documents. This is convenient since packages are the “unit-size” of most of the development efforts.

[Note: Previous versions did not distinguish between documentation for, headers, modules, and test executables. The current presentation has been considerably rearranged; however all material required in previous versions is still required in this version.]

7.3.1 Header file documentation

The documentation for each header file within a package `include/`-directory will form a LaTeX section within the package chapter. All header documentation will have a uniform format and include the following information in this order.

1. **Short description**: Each header section will begin with a short (one sentence) description of the header.
2. **Synopsis**: A somewhat more extensive explanation of the purpose of the header file. Keep in mind, some detailed information may be better left to the documentation of the individual modules and functions that use this header.
3. **Error codes and messages**: The error codes and messages must be auto-extracted and included in the documentation in a LaTeX table. [`laldoc` has a simple way of doing this.] Additional explanation of the errors can go after the table. In particular, explain what measures are taken to handle errors.
4. **Structures**: If you must define a new structures for the input, output, or parameter block for your routine, you must document them here. Note: these structures must be entered in the LaTeX index with an `/index{}` command. The `LAL` prefix on data-structure names should be omitted when putting them in the index.
5. **Author-Id block**: This should appear as a footnote at the bottom of the last page.

7.3.2 Module documentation

The documentation for each module that includes a given header file will form a LaTeX subsection within the header-file section. The documentation for a module will have a uniform format and include the following information in this order.

1. **Short description**: Each module subsection will begin with a short (one sentence) description of the module.
2. **Prototypes**: The prototypes for all the functions in this module must appear here. Note: these functions must be entered in the LaTeX index with an `/index{}` command. The `LAL` prefix should be omitted when putting them in the index.
3. **Description**: Explain how to use the functions. Give detailed information about the arguments. Explain any run-time options that may be invoked. Remember that any non-LAL structures used as arguments should be documented in the header-file section.
4. **Algorithm**: Explanation of the algorithm.

5. **Uses**: A list of all the routines that this module uses.
6. **Notes**: Additional discussion can go here.
7. **Validation Information**: This section is a placeholder for formal results of validation testing. In the mean-time please put information about timing and accuracy here.
8. **Author-Id block**: This should appear as a footnote at the bottom of the last page.

7.3.3 Component-level test documentation

The documentation of the test programs will form a subsection of the header file section. The documentation for the programs will have a uniform format and include the following information in this order.

1. **Short description**: Each test program subsection will begin with a short (one sentence) description of the module, e.g. ‘‘`SampleTest.c` is an executable that tests all functions specified in the header `SampleHeader.h`.’’
2. **Usage**: Show and explain the command line syntax.
3. **Description**: Explain in detail what tests are done and how they work.
4. **Exit Codes**: A LaTeX table containing the exit codes. We strongly suggest that you extract these from the source in the same way error codes are extracted.
5. **Uses**: A list of all the routines that this module uses.
6. **Notes**:
7. **Author-Id block**: This should appear as a footnote at the bottom of the last page.

8 MAINTAINING THE LAL

8.1 Version control for the LAL

The LL and LSC will jointly maintain both the LAL software and the LAL specification. The source code and documentation -- and this document -- will be kept in a CVS repository. When a package is submitted to the library its directory tree will be entered in the CVS repository. The revision history of the files will be available on the web. The LSC Software Coordinator and Software Librarian will over see the day-to-operations of the repository. They will also see that the most up-to-date versions of all code files are publicly --and easily -- available on the web.

8.2 Numbering the LAL releases

In addition to making the individual code pieces available, the LSC Software Coordinator and Software Librarian will periodically issue a “release” of the entire library. The numbering scheme for releases of LAL code will be two numbers 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”. These numbers will be supplied automatically by the CVS. Here X = version number. This is incremented whenever major changes are introduced. If X is incremented, Y is reset to 0. Here 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) modifications for which X is not incremented.

8.3 Validation of LAL code

Verifying that the individual components (functions) work will primarily be the responsibility of the code developers. This is the purpose of the test routines described in **Section 6.2.3**.

The LSC Software Coordinator, the LSC data analysis subgroup chairs and the LL personnel will organize integrated tests of the analysis pipeline through “mock data challenges”. These tests will be conducted to “validate” the code.

8.4. Requesting changes in LAL

LL and LSC will maintain a web page for submitting bug reports and releasing the code. At present this is partially functional: `http://www.lsc-group.phys.uwm.edu/lal/`.

While in the development phase, updating the code and documentation will be largely be the responsibility of the individual code writers. However, as we transition to ‘‘production mode’’, the procedure for updating code will need to be more formal. [During the early stages a-c will apply. In the more formal stage a-e apply.]

a. All modified code will be verified (and validated in a pipeline test if necessary). All affected documentation will be revised to show changes.

b. Once available, a new release will be distributed.

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

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

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

9 DEVELOPMENT TOOLS AND SOFTWARE PACKAGES USED WITH LAL

To keep life simple for the users and developers, we limit the required packages to a few well chosen items. This minimizes the number of learning curves that developers need to be climb before they can start coding, and it limits the number of packages that users need to install before they can use the LAL functions.

9.1 Compiling the LAL

In keeping with the goal of ‘‘broad use” we will try to maintain portability of the LAL, e.g. it currently installs several platforms with several different compilers. This portability may be hard to maintain in the furture, but, as minimum, we will work to insure the LAL compiles and installs on

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

9.2 Development tools:

- GNU CVS: version 1.10 or greater. [Primarily, this will be used by the LSC Software Librarian and Coordinator; other developers shouldn't need this.]
- GNU Autoconf [Primarily, this will be used by the LSC Software Librarian and Coordinator; other developers shouldn't need this.]
- GNU m4: version 1.4 or greater. [Primarily' this will be used by the LSC Software Librarian and Coordinator; other developers shouldn't need this.]
- GNU make: version 3.72 or greater.

LAL software will be delivered with makefiles which, as a minimum, enable installation, compilation and execution of code elements with the hardware and compilers specified above. Each subdirectory in a distribution that contains something to be compiled or installed will come with a file **Makefile.am** file, from which automake will create **Makefile.in**, from which configure will create a **Makefile** in that directory.

9.3 Documentation tools:

- LaTeX
- Custom made automatic documentation tool: `laldoc`.
- PDF (generated by any means).

9.4 Software packages

Currently, FFTW is the only software package required for LAL installation. All others are optional. Let's keep it that way.

- FFTW (**Required**) [FFTW is the current choice for an fft engine; however we have not burned any bridges that would preclude changing to a different package if something better comes along.]
- MPI (**Optional**)
- Frames (**Optional**)
- (C)LAPACK (**Optional**, not implemented yet.)
- Not Numerical Recipes.

APPENDIX A LAL TEMPLATE HEADER FILE

```
/*[Author-Id block must be auto extracted] <lalVerbatim file="LALTemplateHV">
 * Author: Hacker, A. Good
 * $Id: LALTemplate.h$
*** [Note: CVS will always supply file name in the Id.]  </lalVerbatim> ****/

/* A brief (one sentence) description of what this header is for.   */

/* Synopsis and (longer) Description goes  here                        */


#ifndef _LALTEMPLATE_H /* Protect against double-inclusion         */
#define _LALTEMPLATE_H  /* Note the naming convention                 */


#include "LALStdlib.h"  /* Include any other headers                  */


#ifdef __cplusplus     /* Protect against C++ name mangling        */
extern "C" {
#endif


/* You must use  the NRCSID macro to define the RCS ID string       */
NRCSID(LALTEMPLATEH,"$Id: LALTemplate.h$")


/* Define error codes and messages. These must be auto-extracted
 * for inclusion in the documentation
 ********************************** <lalErrTable file="LALTemplateHError"> */

#define LALTEMPLATEH_EONE 1
#define LALTEMPLATEH_ETWO 2

#define LALTEMPLATEH_MSGEONE "An error condition"
#define LALTEMPLATEH_MSGETWO "Another error condition"

/********************************** </lalErrTable> */


/* Define other global constants or macros  (discouraged)          */


/* Define new structures and types. (Use LAL types when possible)  */


/* Include external global variables                               */


/* Declare global function prototypes                              */


void
LALTemplate( LALStatus *stat );


#ifdef __cplusplus
}                   /* Close C++ protection             */
#endif
#endif              /* Close double-include protection  */
```

APPENDIX B LAL TEMPLATE SOURCE FILE

```
/* [Author-Id block must be auto extracted] <lalVerbatim file="LALTemplateCV">
 * Author: Hacker, A. Good
 * $Id: LALTemplate.c$
*** [Note: CVS will always supply file name in the Id.]  </lalVerbatim> ****/

/*
 * The following comments  can (should) form the nucleus of the
 * documentation. However, if the discussion becomes too long
 * and the "code gets lost in the documentation", you MUST move the text
 * elsewhere.  The easiest thing to do is to put it at the end of
 * this module file and /input{} into the documentation here where it is
 * needed.
 *
*/

/* ----------------------------------------------------------------- */

/* A brief description of what the functions in this module do.      */

/* /input{} the file with the extracted function prototypes.         */

/* Description (Describe how to use the functions in this module)    */

/* Algorithm                                                         */

/* Uses  (what other  functions does  this module call)             */

/* Notes (other comments about the code)                             */

/* ----------------------------------------------------------------- */


#include "LALStdlib.h"    /* include headers.order: std, LDAS, LAL   */
#include "LALTemplate.h"  /* include LAL header  for this module     */

/* You must use  the NRCSID macro to define the CVS ID string        */
NRCSID(LALTEMPLATEH,"$Id: LALTemplate.h$")


/*
   Now comes the code:
   [The following order is prefered, but there may be exceptional
   circumstances.]

   1. Constants, enumerated types, structures (used only internally)

   2. Type declarations (used only internally)

   3. Macros.(discouraged)

   4. Extern global variable declarations. (Strongly discouraged!)

   5. Static global variables. (Strongly discouraged!)

   6. Static function declarations:

   7. The functions that make up the guts of this module.
      (Remember to auto-extract the prototypes for inclusion in the
       documentation.)

*/
```