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

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

<table>
<thead>
<tr>
<th>Document Type</th>
<th>LIGO-T010003-00-E</th>
<th>2003/10/09</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAL-LDAS Interface Coding Specification</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Kent Blackburn, Patrick Brady, Duncan Brown, Jolien Creighton, Albert Lazzarini and Alan Wiseman, others?

Distribution of this draft:
Contents

1 Introduction 3
   1.1 The scope of this specification ........................................ 3
   1.2 Applicability .................................................................. 3
   1.3 The underlying design criteria for the interface .................. 4

2 How does LAL fit into LDAS 5

3 Definition of the interface: the functions and the argument datatypes 6
   3.1 Function LALInitSearch() .................................................. 7
   3.2 Function LALConditionData() ............................................ 9
   3.3 Function LALApplySearch() .............................................. 11
   3.4 Function LALFinalizeSearch() ........................................... 13

4 LALWrapper code Organization 14
   4.1 Organization of contributed analysis codes ......................... 14

5 The rules for contributed analysis code in LALWrapper 15
   5.1 The Rules (and the reasons for the rules) for Real-Time Computing in the LDAS 15

6 LALWrapper code documentation 16

7 Maintaining the LALWrapper 16
   7.1 Version control for LALWrapper ........................................ 16
   7.2 Numbering the LALWrapper releases ................................ 16
   7.3 Validation of LALWrapper code ....................................... 16
   7.4 Requesting changes in LALWrapper .................................. 17

8 Development tools and software packages required for shared object development 17
1 Introduction

The LIGO Laboratory is building a modular data analysis pipeline called LDAS (LIGO Data Analysis System). Members of the LIGO Scientific Collaboration (and others) are writing LAL search code to analyze the data. This document describes the requirements on data analysis software intended for execution on the LDAS Beowulf clusters.

The interface between LDAS and LAL is a c++ executable called the wrapperAPI. In order to perform a particular search the wrapperAPI dynamically loads and calls four LAL functions that contain the search code. This document describes these LAL functions and lays out requirements on search code operating in the real-time LDAS analysis environment.

In brief summary, the logic of a search code must be encapsulated in these four functions which form a shared object library:

- LALInitSearch()
- LALConditionData()
- LALApplySearch()
- LALFinalizeSearch()

The wrapperAPI calls each of these functions in the order shown; it also has a built-in loop that repeatedly calls LALApplySearch() until the search is complete.

Since the wrapperAPI provides management and control structure for all parallel data analysis, the arguments of these functions are generic data type of sufficient flexibility to encompass the needs of all search codes. This document describes the arguments of these functions, i.e how LDAS will get the data to your analysis code and how your analysis code will pass the results back to LDAS.

1.1 The scope of this specification

This document formally defines the LAL functions that fit into the LDAS-LAL interface. This is not a “how-to” manual for writing search code; however it is a “how it works document”. Most importantly, this document spells out how information must flow from LDAS and wrapperAPI to the LAL search code.

1.2 Applicability

The LAL Specification states “all participating groups will be required to analyze LIGO data using LAL-compliant software. However, at present, there is no similar requirement that states all groups will be required to analyze data in the LDAS environment. In spite of the lack of a formal rule, the LSC and LIGO Laboratory encourage developers to develop their code in such a way that it will fit into LDAS through the wrapperAPI. It must be emphasized that the wrapperAPI design does not impose significant constraints on algorithm design, although the need for reliable real-time analysis does impose certain reporting requirements.

\[\text{Some of the material in this document is redundant with the wrapperAPI baseline requirement document. If there is any discrepancy, that document takes precedence.}\]
1.3 The underlying design criteria for the interface

The requirements presented here strike a delicate balance between two competing design criteria. On the one hand, one wishes to keep the LDAS-LAL interface simple and flexible. A simple, non-restrictive interface does not limit the types of algorithms that can be used within LDAS and will not limit the number of developers willing to climb the learning curve to fit their code into the slot. On the other hand, LDAS is designed to run multiple searches simultaneously, often analyzing data in real-time as it is acquired. This computing environment requires a strict rules for the search-code writers. The design of the wrapperAPI is a compromise between these two competing criteria.

The current interface gives developers complete control the computational and algorithmic aspects of their data analysis codes; the bulk of their code would be the same if they were writing stand-alone code. The two important differences between writing code for the LDAS environment and writing stand-alone code are:

- Developers are not responsible for reading data from frames within LDAS. Data flow is handled by LDAS and controlled by the developer via a user command and TCL script to execute it. This control layer is beyond the scope of this document.

- Developers are responsible for providing status and progress of the analysis at run-time – the mechanism is described below. In concert with the reporting requirements, code must support some management initiated and executed by the wrapperAPI.

Since the LDAS-LAL interface is flexible, it is hoped that data analysis code writers will install LDAS at their home institutions and do their code development within the LDAS environment. At the very least, the initial development of code to run under the wrapperAPI should be done using a standalone version installed on your local machine.
2 How does LAL fit into LDAS
3 Definition of the interface: the functions and the argument datatypes

In this section we explain how to the data and parameters get into your search code, and how you get the results out. Developers should be aware of one over-arching rule for their code: *All four functions must obey LAL coding standards as indicated in the LAL software specification.* The functions in each analysis code are LAL function, therefore, to the extent possible, they should adhere to the rules stated in Section 5 of the LAL Spec. Deviations from these rules are indicated below.

The interface to all search codes is the same by design. For this reason, most of the arguments are generic and do not change datatype from search to search. The exception is an argument of type `void` in each of the functions. The developer will define a structure at the end of this pointer in the search header file. This is the structure used to port information between the four search functions. In the discussion below, the variable name assigned to the search-specific structure is `searchParams` and the data structure is (which the user must define) will be called `MySearchStruct`.

---

2In `LALApplySearch()` the `void *` is hidden inside the struct `LALApplySearchParams`. 
3.1 Function LALInitSearch()

This functions is initiated on all nodes except the wrapperMaster. It is intended to parse the command line parameters (char *argv[]), and prepare them to be passed to the other functions. It should also be used to allocate memory for the search-specific data structures.

Prototype

```c
void LALInitSearch(  
    LALStatus *status,  
    void **searchParams,  
    LALInitSearchParams *initSearchParams
);
```

Description

**LALStatus *status**: This is the standard LAL status structure. It is rigorously defined in the LAL Specification. Since this is a pure LAL function, the first argument is LALStatus.

**void **searchParams**: This void ** is a handle for your search-specific data structure, say MySearchStruct. Its purpose is to carry (pointers to) search-specific information (e.g. the data and the command line arguments) to your other functions. You must typedef this structure in your InitSearch.h header file, and attach it to the void ** in LALInitSearch(), e.g.

```c
*searchParams = LALMalloc( sizeof(MySearchStruct));
```

**LALInitSearchParams *initSearchParams**: The structure LALInitSearchParams is typedefed to the InitParams in LALWrapperInterface.h. The values in this data structure are prepared by the wrapperAPI and passed to LALInitSearch(); this structure definition cannot be changed in contributed analysis code. The elements of InitParams are:

- **initSearchParams -> argc** is the argument count
- **initSearchParams -> argv[]** is the character string containing the command line filter parameters. A valid parameter string must begin with *initSearchParams->argv[0]="-filterparams" and will be followed by the search specific parameters, e.g. the string might look like -filterparams (1,2,3,(4,5),6.7,8.9). The wrapperAPI baseline requirements document give requirements for sanity checks of the parameters.
- **initSearchParams -> startTime** is the start time of the wrapperAPI job in seconds since 0h 1 January 1970.
- **initSearchParams -> dataDuration** is the total amount of data requested for this job rounded up to the nearest second.
- **initSearchParams -> realtimeRatio** multiplied by dataDuration gives the maximum time alloted to this wrapperAPI job. If the analysis is not completed in this time, the job may be terminated by the mpiAPI.
initSearchParams -> rank indicates the node on which LALInitSearch was called. A value −1 indicates the wrapperMaster, 0 indicates a searchSlave, and 1 indicates the searchMaster.

In order to make use of the filterParameters information, LALInitSearch() parses argv and passes the information to the other functions using the “search specific” output structure that is attached to the void **.

Uses

Notes
3.2 Function **LALConditionData()**

This function is initiated on all nodes except the wrapperMaster. It is intended to perform search-specific data conditioning. If the run-time wrapperAPI option is set to `-dataDistributor=C`, then LALConditionData() will take responsibility for distribution of the data to the slaves with MPI communication. See note below. In general, LALConditionData() will unpack the input data from the generic *inout structure and insert it into the appropriate LAL data types.

**Prototype**

```c
void
LALConditionData(
    LALStatus *status,
    LALSearchInput *inout,
    void *searchParams,
    LALMPIParams *mpiParams
);
```

**Description**

**LALStatus *status**: This is the standard LAL status structure. It is rigorously defined in the LAL Specification. Since this is a pure LAL function, the first argument is LALStatus.

**LALSearchInput *inout**: This structure will contain your input data. It is typedefed in LALWrapperInterface.h to be type inPut which is defined in wrapperInterface-Datatypes.h. The elements of inPut are:

- **inout -> numberSequences** is the number of input sequences requested by the search code and packed into the multiDimData structure.

- **inout -> stateVector** TBD.

- **inout -> multiDimData** contains the input data. It is rigorously defined in the “Wrapper API’s Baseline Requirements” [http://www.ligo.caltech.edu/docs/T/T990097-15.pdf](http://www.ligo.caltech.edu/docs/T/T990097-15.pdf). You will need to unpack the multiDimData and repack it using LAL structures.

**void *searchParams**: This void * is a pointer to your search-specific data structure which you typedefed in InitSearch.h. Remember to recast it before you try to address its elements, i.e, `localParams = (MySearchStruct *) searchParams` ; This structure will contain the command line arguments that you parsed and attached in LALInitSearch. You should also attach the data you unpack from the multiDimData, as this structure will also be passed to LALApplySearch().

**LALMPIParams *mpiParams**: This is a pointer to the MPI parameters including the MPI communicator, which can be used to do MPI communication within the LALConditionData() routine. **Use of this argument is strongly discouraged: all MPI communication should be restricted to the LALApplySearch() function.** The MPI communicator can also be used to determine the
number of nodes available using the `MPI_Comm_size()` function. Doing this is also discouraged as the result will be incorrect if the number of nodes available changes during execution. It is best to ignore this field altogether and do everything involving the MPI communicator in the `LALApplySearch()` function.

Uses

Notes
3.3 Function **LALApplySearch()**

This function is initiated on all nodes except the wrapperMaster. It should contain the computational engine of the analysis code. It is executed repeatedly (i.e inside a loop) by the wrapper-API. On the searchMaster, it should return often to provide progress to the wrapperAPI and receive instructions from the mpiAPI. See Section 5.1 for a complete set of requirements.

**Prototype**

```c
void LALApplySearch(
    LALStatus *status,
    LALSearchOutput *output,
    LALSearchInput *input,
    LALApplySearchParams *params
);
```

**Description**

**LALStatus ** *status**: This is the standard LAL status structure. It is rigorously defined in the LAL Specification. Since this is a pure LAL function, the first argument is LALStatus.

**LALSearchOutput ** *output**: This is the structure where you put your results. It is typedefed in LALWrapperInterface.h to be type SearchOutput which is defined in wrapperInterfaceDatatypes.h. The elements of SearchOutput are:

```c
typedef struct
{
    INT4 numOutput;
    outPut* result;
    REAL4 fracRemaining;
    BOOLEAN notFinished;
}SearchOutput;
```

The elements of SearchOutput are:

- **output -> numOutput** is the number of events that are packed into result.

- **output -> result** is the structure into which you pack your event information. The type outPut is defined in the “Wrapper API’s Baseline Requirements” http://www.ligo.caltech.edu/docs/T/T990097-15.pdf.

- **output -> fracRemaining** fraction of analysis remaining to be done when LALAppySearch returns; remember, this function is called many times. Developers should insure that they have a routine which computes this number as accurately as possible. This need only be set on the searchMaster.

- **output -> notfinished** indicates that the analysis is complete if set to FALSE, i.e. LALAppySearch should not be called again on the node which returned this value. Otherwise it should be set to TRUE on any node that needs LALAppySearch called again.
**LALSearchInput** *input*: Same as LALSearchInput structure described above. For some simple searches, there may be no need for data conditioning, and therefore the unpacking of the input data can be postponed until LALApplySearch() is called.

**LALApplySearchParams** *params*: This contains the search specific information. It is defined in LALWrapperInterface.h

```c
typedef struct
tagLALApplySearchParams
{
    LALMPIParams *mpiParams;
    void *searchParams;
}
LALApplySearchParams;
```

The elements of LALApplySearchParams are:

- **params -> mpiParams** provide the MPI_Comm and some wrapperAPI-centric information. LALMPIParams is typedefed to SearchParams in LALWrapperInterface.h; in wrapperInterfaceDatatypes.h SearchParams is typedefed to be

```c
typedef struct
{
    MPI_Comm* comm; /* wrapper slave COMM_WORLD */
    MPIapiAction* action; /* instruction from mpiAPI to search code */
}SearchParams;
```

- **params -> mpiParams -> comm** the MPI communicator containing information about the searchMaster and searchSlave nodes.

- **params -> mpiParams -> action** is needed when the mpiAPI will execute dynamic load-balancing. It is typedefed to be

```c
typedef struct
{
    INT4 add;     /* number of nodes added or subtracted */
    BOOLEAN mpiAPIio; /* command from mpiAPI: false - exit, true - continue */
}MPIapiAction;
```

where add is the number of nodes added to, or subtracted from, the communicator when load-balancing was executed. The BOOLEAN mpiAPIio is a flag which indicates if the node will continue, or that the code has been instructed to exit by the mpiAPI.

- **params -> searchParams** is the void * pointer to the same search-specific structure discussed for the previous functions. As before, it should be recast, i.e, localParams = (MySearchStruct *) searchParams ;.

**Uses**

**Notes**
3.4 Function \texttt{LALFinalizeSearch()}

Frees the memory allocated for the search-specific datatype \texttt{**searchParams}.

Prototype

\begin{verbatim}
void LALFinalizeSearch(
    LALStatus   *status,
    void        **searchParams
);
\end{verbatim}

Description

\textbf{LALStatus *status}: This is the standard LAL status structure. It is rigorously defined in the LAL Specification. Since this is a pure LAL function, the first argument is \texttt{LALStatus}.

\textbf{void **searchParams}: This \texttt{void **} is a handle for your search-specific data structure, say \texttt{MySearchStruct}.

Notes

Need to briefly explain the two modes of data distribution.
4 LALWrapper code Organization

This section explains the layout of code within the LALWrapper. 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. 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.

Note that the source code that makes up the shared object Libraries are genuine LAL functions, and therefore the requirements for code organization are the those given in the Chapter 6 of LALspec. There is currently one exception: a /test directory is not required in LALWrapper since that would require a standalone wrapperAPI to be installed locally. (This rule may change as the code matures.) In the LAL distribution this directory contains the executables that drive the routines for testing purposes.

4.1 Organization of contributed analysis codes

All LALWrapper components will reside in a single directory, called lalwrapper, and its subdirectories. The LSC Software Coordinator and Software Librarian will maintain an official master copy of the LALWrapper source in the CVS repository. An official release will be distributed as a tar-ball thru the LAL web pages. Users can download and install a release on their own machines. Within this top level directory, their will be a subdirectory (lalwrapper/contrib/) where the analysis codes will reside. Within this subdirectory, every analysis code will have a named directory that contains all files associated with that analysis code (e.g. lalwrapper/contrib/inspiral). The development of contributed analysis code will be the primary way collaborators will contribute to the LALWrapper. A contributed analysis code (e.g. lalwrapper/contrib/mysearch) should have the source files, documentation and Makefiles in the following subdirectories:

- lalwrapper/contrib/mysearch/include: All the header files associated with mysearch. The header file MySearch.h should define the MySearchStruct type which is needed by the four search functions. Each of the four required functions should have associated header files (e.g. MySearchInitSearch.h, MySearchConditionData.h, etc). Other header files can be present at the discretion of the developer, however, all header files must conform to the format and style described in Section 6.2.1 of the LAL specification.

- lalwrapper/contrib/mysearch/src: all the source files associated with the component. Each of the four required functions should have associated source files (e.g. MySearchInitSearch.c, MySearchConditionData.c, etc). Other source files can be present at the discretion of the developer, however, all source files must conform to the format and style described in Section 6.2.2 of the LAL specification. In particular, each source file must be associated with one (and only one) header file.

- lalwrapper/contrib/mysearch/test: test scripts and all supporting files associated with component-level tests. The format and style of these tests remains TBD as LAL, LALWrapper and LDAS mature.
• **lalwrapper/contrib/mysearch/doc**: There will be a \texttt{LATEX} file in this directory capable of assembling stand-alone documentation for this package. There will also be a \texttt{LATEX} file that forms a chapter in comprehensive manual for the entire LALWrapper. Before auto-extraction with \texttt{laldoc}, much of the text source for the documentation may reside in the code files. See Section 7 of the LAL specification for details.

## 5 The rules for contributed analysis code in **LALWrapper**

Contributed analysis codes must meet certain standards before they will be admitted to the LDAS real-time computing environment. These rules must be met by all analysis codes in LALWrapper in production releases. As LALWrapper evolves from development to production software, more of these rules will be gradually enforced. In this section, we indicate which rules must be obeyed by contributed code at this time. If the upcoming release is greater than or equal to the release number listed with each requirement, then analysis code must adhere to that rule.

### 5.1 The Rules (and the reasons for the rules) for Real-Time Computing in the LDAS

When running analysis alongside other analyses in real-time it is not enough to simply conform to the interface specification in the previous sections. The real-time LDAS computing environment is high-speed, communal computing in close quarters: there must be strict rules. In this section, we present a set of requirements and restrictions for search codes running within this environment.

1. Each type of analysis must be executed as a sequence of short jobs which can be completed in a short time, i.e 1–few hours. For example, a search may bite off an hour’s worth of data, analyze it, terminate, and then restart with the next hour’s worth of data.

   • Reason 1: To avoid job failure due to secular memory leaks.
   • Reason 2. Load balancing. Although there will be means to reassign nodes to running jobs, short execution times allow the possibility to redistribute resources in a timely manner.

2. Progress through the analysis must be periodically reported to the \texttt{wrapperAPI}. To achieve this, the search function \texttt{LALApplySearch()} is called by the \texttt{wrapperAPI} in a while loop that ends when \texttt{notFinished} is set to \texttt{FALSE}. \texttt{LALApplySearch()} should return at least 10 times per job (or every \texttt{\sim} 5 minutes on all nodes. A value between zero and unity should be assigned to \texttt{fracRemaining} on the \texttt{wrapperMaster} each time \texttt{LALApplySearch()} returns. fraction remaining.

   • Reason 1: The wrapper may need to make load balancing decisions while the job is still running.
   • Reason 2: If a job hangs, and doesn’t report for a long time the LDAS job management system needs to be able to figure it out and kill the job.
6 LALWrapper code documentation

All the functions written by developers should be LAL functions, and therefore the philosophy, the requirements and the organization of the documentation are those given in the Chapter 7 of LAL Specification. In particular, the same auto-documentation tool, laldoc will be used.

In summary, the documentation requirements are:

- Documentation will be written in LaTeX.
- The author and CVS Id block in the code must be auto-extracted from the code and automatically included in the documentation.
- Error codes and error descriptions must be auto-extracted from the header files and automatically included in the documentation.
- Function prototypes must be auto-extracted and included in the documentation.
- All functions must be entered into the LaTeX document index with an \index{} command.
- All non-LAL data structures must be entered into the LaTeX document index with an \index{} command.
- Do not let the code get lost in the documentation.

7 Maintaining the LALWrapper

7.1 Version control for LALWrapper

The LL and LSC will jointly maintain both the LALWrapper software and this 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 oversee the day-to-day operations of the repository. They will also see that the most up-to-date versions of all code files are publicly available on the web.

7.2 Numbering the LALWrapper releases

The numbering of versions and releases of the library will done according to the same rules as the LAL specification.

7.3 Validation of LALWrapper code

Verifying that the individual components (functions) work will primarily be the responsibility of the code developers. 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 in LALWrapper. A complete record of these challenges will be maintained in a CVS repository which collects together a final report,
test checklists completed during the MDC, scripts and test programs, input data, output data and correct results. Each MDC should result in a set of tests which can be used to validate later releases of the analysis codes.

7.4 Requesting changes in LALWrapper

The LIGO Laboratory and LSC will maintain a web page for submitting bug reports and releasing the code. Currently, this can be found at [http://www.lsc-group.phys.uwm.edu/lal](http://www.lsc-group.phys.uwm.edu/lal).

While in the development phase, updates to code and documentation will be the responsibility of the developers. However, as we transition to production releases, the procedure for updating code will become more formal. [During the early stages a-c will apply. In the more formal stage a-e apply.]

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

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

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

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

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

8 Development tools and software packages required for shared object development

As the LALWrapper shared object library is a library of LAL functions, the development tools should be the same as those required for LAL, e.g. GNU CVS, Linux [Redhat 6.0 or greater], gcc, etc. In addition, some parts of the LDAS may be required.