

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

Document Type	LIGO-T01XXXXXX- Z	2004/02/17
How to develop search code to run in LDAS under wrapperAPI		
MPI working group		

Distribution of this draft:

LSC and LIGO

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/>

Contents

1	Introduction	3
1.1	LDAS	3
1.2	LAL	4
1.3	LALwrapper	4
2	Getting and installing the software	6
2.1	Installation of required development tools	6
2.2	Configuring your environment	6
2.3	LAL	7
2.4	LALwrapper	8
2.5	wrapperAPI	8
3	Running your first wrapperAPI job	10
3.1	LAM schema files	10
3.2	wrapperAPI input files, e.g. wrapper.ilwd	11
3.3	wrapperAPI output files	11
4	Adding helloworld to LALwrapper	13
4.1	HelloWorld.h	17
4.2	HelloWorld.c	17
4.3	helloworld.tex	18

1 Introduction

The preferred method of developing code to be executed by the `wrapperAPI` is to install LAL, LALwrapper and a standalone `wrapperAPI` on your system and your shared object within the LALwrapper source tree. This allows the `configure` script to generate the necessary makefiles with the appropriate flags.

Before we begin with the technical aspects of development, a brief reminder of the software model adopted by the LSC may be useful. There are three software components: (i) the LIGO data analysis system (LDAS), (ii) the LIGO/LSC Algorithm Library (LAL) and (iii) the LAL-LDAS interface and search code library (LALwrapper).

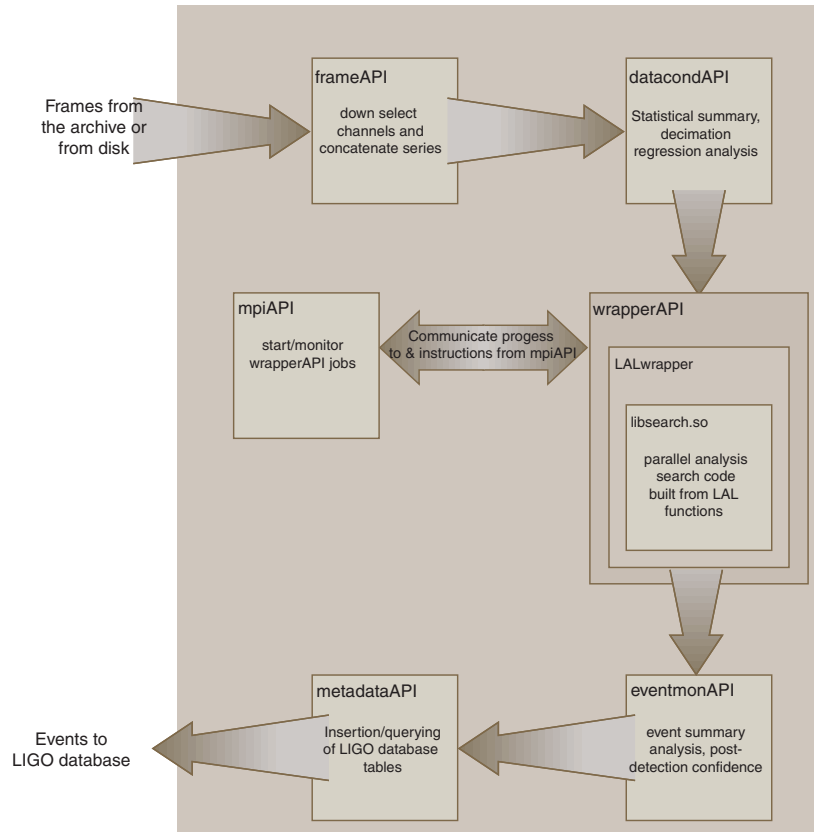


Figure 1: This figure illustrates a simple example data flow through LDAS. LALwrapper and LAL are shown inside the wrapperAPI.

1.1 LDAS

This is the exoskeleton which supports the internal organs of the LSC search codes. It is developed and maintained by the LIGO Laboratory. It is broken into modules – the technical term for these modules is an Application Programmer Interface (API) – which provide different pieces of functionality required by a search pipeline. For example, the frame API can read frames from disk or archive, extract information requested by the user, concatenate the requisite time series as necessary and send the information to the next API. In general, the LDAS API's adopt a model in which they receive data through an inflow (a data socket), apply methods to the data internally, and then pass the data to the next API through some outflow (another

data socket). This model means that the various API's can be assembled into many different topologies which allow complex and difficult data analysis tasks to be carried out. There is a standard method for logging information from each API and a special API which allows the user to monitor and control jobs (the Control Monitor API). Thus, LDAS provides a standardized infrastructure for accessing and manipulating the data, and for keeping detailed records of the analysis. Scientific search codes slot into LDAS via the wrapperAPI [1, 2]. The search codes are *not* part of LDAS software.

1.2 LAL

This library contains the numerical algorithms used to construct the gravitational wave search codes. It is developed and maintained by the LSC. The LAL specification determines the data types, calling format, and other coding requirements for functions in this library. All scientific searches will be executed using this library [3]. The functions in LAL are like cells. Each function can do something (relatively) simple. When many functions are put together in an appropriate manner, they can execute complex activities like searching for waves from coalescing compact binaries.

1.3 LALwrapper

This is a set of libraries. It is written and maintained by the LSC. The LAL specification governs coding practice in the search codes implemented through this package [3]. A library might execute a type of search, e.g. a search for waves from coalescing compact binaries, or some other computationally intensive data manipulation. Each library contains five functions which are called by the wrapperAPI: `initSearch()`, `conditionData()`, `applySearch()`, `freeOutput()`, and `finalizeSearch()`. Developers do not deal directly with these five functions, instead they deal with four LAL compliant counterparts: `LALInitSearch()`, `LALConditionData()`, `LALApplySearch()`, and `LALFinalizeSearch()`. The `freeOutput()` function is the same in all search codes; it is needed only because wrapperAPI cannot free memory allocated using `LALMalloc()`. A detailed specification of these functions and their requirements can be found in [1]. `LALApplySearch()` is called in a loop on all search nodes (see Fig. 2). It executes the search algorithm. A few points about `LALApplySearch()`:

- The MPI communicator is passed to this function. This allows for truly parallel implementations of search algorithms to be coded libraries since MPI processes on any of the search nodes can, in principle, communicate with others.
- This function must return at least 10 times during a search, see Ref. [1]. On the search master node, it may return more often. For example, in a slave driven search code it might return each time that a slave node sends results to the search master.
- Upon return, the boolean flag `output->notFinished` must be set; this must be done on all nodes not just the search master. `FALSE` indicates that `applysearch` is finished on that node. On the search master, `output->notFinished` should not be set to `FALSE` until all search slave nodes have finished. In addition, on the search master `output->fracRemaining` must be set to a number between 0 and 1 when `LALApplySearch()` returns. This number indicates the progress of the job by informing the wrapperAPI what fraction of the job remains to be executed. This information is used by the wrapperAPI and `mpiAPI` to track the progress of jobs, so it should be estimated as accurately as possible. (N.B. The wrapperAPI can be run in a *load balancing* configuration which requires further constraints on the execution `output->fracRemaining` and coordination on the search nodes. See Refs. [1, 2] for details.)

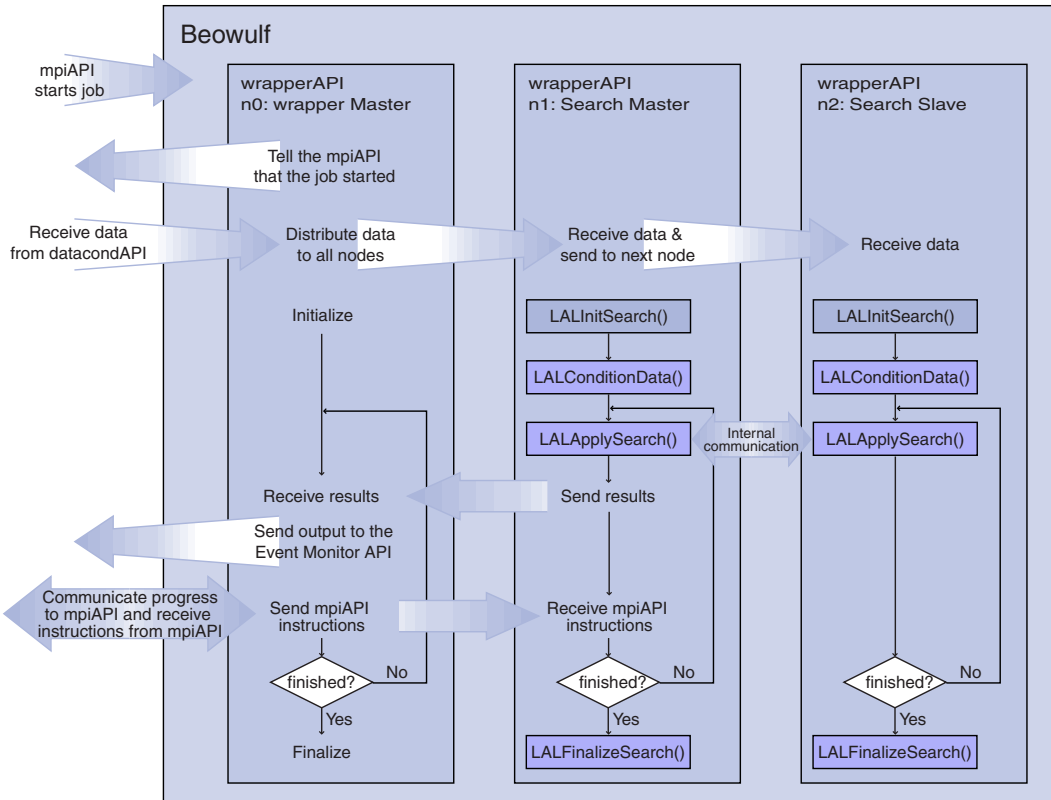


Figure 2: This figure illustrates the logic of the wrapperAPI with respect to the four search specific functions loaded from the LALwrapper shared object library. This wrapperAPI job uses three nodes: n0, n1, n2. If the job used more than this, the logic and communication of nodes n3–nN would be the same as n2. Note the naming convention used above. The *wrapper master* is always n0; it is responsible for broadcasting the data (this is the default behavior, see the wrapperAPI baseline requirements for other options), sending/receiving job information to/from the mpiAPI, buffering results from the search code and sending them to the eventmonAPI. Scientific search code is not executed on the wrapper master. The search is executed on nodes n1–nN. The *search master* is always n1. It is the master process associated with any parallel search algorithm. The *search slaves* are nodes n2–nN.

2 Getting and installing the software

We describe the installation of LAL, LALWrapper and the components of LDAS needed to run the wrapperAPI in standalone mode. Installing the software needed by LDAS takes a little time and effort. Once the infrastructure is in place, however, it is straightforward to develop search code to run under the wrapperAPI. Ultimately, testing should be done on a system running LDAS but a standalone wrapperAPI implementation is preferable during the first stage of development.

2.1 Installation of required development tools

LAL, LALWrapper and LDAS development takes place in a standardized environment of software tools. This avoids confusion caused by building with different tools and in different environments. The LDAS/LAL development tools are required to reside in a directory hierarchy available at `/ldcg`. (This directory may be a link to another, somewhere else on your system; however, these packages *must* be available through this reference.)

Detailed instructions describing the packages required for the stand alone wrapperAPI in `/ldcg`, their version numbers, where to obtain them, how to patch them (if necessary), and how to install them is maintained on the LDAS web page. Go to and follow the instructions in Sec. 1 of <http://ldas-sw.ligo.caltech.edu/> to install the packages. *Follow only the instructions in Sec. 1; do not retrieve the source as described in Sec. 2.* You will need root permission on your system to carry out this installation process. Be sure to change the environment variables as explained in the LDAS install document.

2.2 Configuring your environment

While developing, we recommend that you install the LAL, LALWrapper and wrapperAPI somewhere under your home directory. If you follow the instructions below, the LAL, LALwrapper and LDAS libraries will be in `$LALPREFIX/lib`; the documentation in `$LALPREFIX/doc`; the header files in `$LALPREFIX/include`; the binaries in `$LALPREFIX/bin`. The environment `LALPREFIX` should be set to the absolute path where you want to install these things. For example, user `patrick` might put the source code in `/home/patrick/src` and set `LALPREFIX` to `/home/patrick`.

You will need to choose a convenient location for the wrapperAPI resource file. It is acceptable to put it in the `LALPREFIX` directory. If you put it somewhere else, edit the `WRAPPER_RESOURCE_FILE` environment variable below.

To make the appropriate binaries accesible to you, check that your path is set correctly. It is also useful to add environment variables for the CVS servers. If you have a `lal` or `lalwrapper` CVS login, change `anonymous@gravity.phys.uwm.edu` to `your_user_name@gravity.phys.uwm.edu` in the environment variables below. If you are using `bash`, add the following lines to your `.bash_profile` file

```
export LALPREFIX=${HOME}           # <---- Change this as appropriate
export PATH=${LALPREFIX}/bin:/ldcg/bin:${PATH}
export WRAPPER_RESOURCE_FILE=${LALPREFIX}/share
export LAMBHOST=${LALPREFIX}/share/lam-bhost.def
export LALCVS=":pserver:anonymous@gravity.phys.uwm.edu:/usr/local/cvs/lal"
export LALWRAPPERCVS=":pserver:anonymous@gravity.phys.uwm.edu:/usr/local/cvs/lalwrapper"
export LDASCVS=":pserver:readonly@ldas-sw.ligo.caltech.edu:/ldcg_server/common/repository"
```

If you are using `csh` or a derivative, add the following lines to your `.cshrc` file

```
setenv LALPREFIX $HOME
setenv PATH $LALPREFIX/bin:/ldcg/bin:$PATH
setenv WRAPPER_RESOURCE_FILE ${LALPREFIX}/share
```

```
setenv LAMBHOST ${LALPREFIX}/share/lam-bhost.def
setenv LALCVS ":pserver:anonymous@gravity.phys.uwm.edu:/usr/local/cvs/lal"
setenv LALWRAPPERCVS ":pserver:anonymous@gravity.phys.uwm.edu:/usr/local/cvs/lalwrapper"
setenv LDASCVS ":pserver:readonly@ldas-sw.ligo.caltech.edu:/ldcg_server/common/repository"
```

These environment variables must be set before running `lamboot`, so it is a good idea to log out and log back in again before continuing. **Note:** With `libtool` version 1.4.2, or greater, there is no need to have the `LD_LIBRARY_PATH` environment variable set. Library paths are hard coded into the shared objects using the `-rpath` compiler option. This is the correct way to do this, using `LD_LIBRARY_PATH` is incorrect.

2.3 LAL

In the following commands, remember `LALPREFIX` is the absolute directory path where you want to install the LAL library. If `LALPREFIX` does not exist, you must create it:

```
mkdir $LALPREFIX
```

Then create the directory into which you wish to put the source files for LAL, LALWrapper and LDAS:

```
mkdir $LALPREFIX/src
```

The LAL software is maintained in a CVS repository – CVS stands for Concurrent Version-control System which is tool to allow multiple developers to manipulate the same software, meerging differences and identifying conflicts between changes if they arise. Obtain the LAL package from the CVS repository as follows:

```
cd $LALPREFIX/src
cvs -d $LALCVS login
```

At this point, you will be asked for a password. The password for the anonymous user is `lal`. If you have a your own username, use the password that you have been given. The version of LAL that is checked-out is identified by the argument to the `-r` option in the commands below. The `HEAD` tag checks out the current development version. If you want to check out a released version, replace the `HEAD` tag by `release-X-Y` where `X` and `Y` are integers which identify the release version number as `X.Y`.

```
cvs -d $LALCVS checkout -rHEAD lal
```

You have now obtained the latest development version of LAL.

To build and install the LAL software suite,

```
cd $LALPREFIX/src/lal
./00boot
./configure --prefix=$LALPREFIX --disable-static --enable-mpi \
  --with-extra-cppflags="-I/ldcg/include" \
  --with-extra-cflags="-fexceptions" \
  --with-extra-ldflags="-L/ldcg/lib"
make
make check
make dvi
make install prefix=$LALPREFIX/stow_pkgs/lal-howto
cd $LALPREFIX/stow_pkgs
stow lal-howto
```

This completes the installation and testing of LAL.

2.4 LALwrapper

Obtain the LALwrapper package from the the CVS repository as follows:

```
cd $LALPREFIX/src
cvs -d $LALWRAPPERCVS login
```

The password for the anonymous user is `lalwrapper`. The version of LALWrapper that is checked-out is identified by the argument to the `-r` option in the commands below. The `HEAD` tag checks out the current development version. If you want to check out a released version, replace the `HEAD` tag by `release-X-Y` where `X` and `Y` are integers which identify the release version number as `X.Y`.

```
cvs -d $LALWRAPPERCVS checkout -rHEAD lalwrapper
```

You have now obtained the latest development version of LALWrapper.

To build and install the LALWrapper software suite,

```
cd $LALPREFIX/src/lalwrapper
./00boot
./configure --prefix=$LALPREFIX \
  --with-extra-cppflags="-I$LALPREFIX/include -I/ldcg/include" \
  --with-extra-ldflags="-L$LALPREFIX/lib -L/ldcg/lib"
make
make check
make dvi
make install prefix=$LALPREFIX/stow_pkgs/lalwrapper-howto
cd $LALPREFIX/stow_pkgs
stow lalwrapper-howto
```

This completes the installation of LALwrapper.

2.5 wrapperAPI

To build the `wrapperAPI` in standalone mode requires a subset of LDAS components to be downloaded. The easiest way to get these is through the CVS repository. There is a `readonly` user; contact Kent Blackburn or Albert Lazzerini to obtain the password. The following commands also checkout the latest CVS version of the `wrapperAPI` and the components of LDAS that it needs. The version of LDAS that is checked-out is identified by the argument to the `-r` option in the commands below. The `HEAD` tag checks out the current development version. If you want to check out a released version, replace the `HEAD` tag by `ldas-X_Y_Z` where `X`, `Y`, `Z` are integers which identify the release version number as `X.Y.Z`.

Log into the repository and download the necessary files as follows:

```
cd $LALPREFIX/src
cvs -d $LDASCVS login
cvs -d $LDASCVS checkout -rHEAD -l ldas
cvs -d $LDASCVS checkout -rHEAD -l ldas/lib
cvs -d $LDASCVS checkout -rHEAD ldas/lib/perceps
cvs -d $LDASCVS checkout -rHEAD ldas/lib/general
cvs -d $LDASCVS checkout -rHEAD ldas/lib/ilwd
cvs -d $LDASCVS checkout -rHEAD ldas/lib/dbaccess
cvs -d $LDASCVS checkout -rHEAD ldas/dbms
cvs -d $LDASCVS checkout -rHEAD -l ldas/api
cvs -d $LDASCVS checkout -rHEAD ldas/api/wrapperAPI
cvs -d $LDASCVS checkout -rHEAD ldas/api/genericAPI
cvs -d $LDASCVS checkout -rHEAD ldas/api/test
```

Note that some commands involve the `-l` option and others do not. The distinction is important: do not lose it.

To build LDAS, issue the commands

```
cd $LALPREFIX/src/ldas
./build-ldas --prefix=$LALPREFIX --disable-metadata-api
```

The `build-ldas` command will take some time to complete (on order 10m on an unloaded, relatively modern system). When the build starts, a subdirectory will be created into which all the programs, libraries and log files are put. The directory name follows the particular system on which you compile, e.g. on a Pentium III running Linux, the directory would be called `Linux-i686`. For simplicity, I use this name below but it may be different on your system.

You can check the progress of the build as follows. Using another terminal window, `cd` into the directory `$LALPREFIX/src/ldas/Linux-i686/build_logs`. This directory will contain several files which are created as different phases of the build proceed. If the build completes, there should be no error messages in the file `make`.

Once the build completes, finish the installation process as follows (remember to replace `Linux-i686` with the appropriate directory name):

```
cd $LALPREFIX/src/ldas/Linux-i686
make install
cd $LALPREFIX/stow_pkgs
stow ldas-X.Y.Z
```

where X, Y, Z are integers which identify the version number.

Now copy the `LDASwrapper.rsc` resource file to the location specified in the `WRAPPER_RESOURCE_FILE` environment variable.

```
cp $LALPREFIX/bin/LDASwrapper.rsc $WRAPPER_RESOURCE_FILE
```

Edit the `LDASwrapper.rsc` resource file as follows:

1. To disable communication with the `mpiAPI`, change the `enable_mpiAPI` from `TRUE` to `FALSE`.
2. To disable communication with the `resultAPI`, change the `enable_resultAPI` lines from `TRUE` to `FALSE`.
3. `wrapperAPI` uses `dump_data_directory` (default value is `/ldas_outgoing/jobs`) when `enable_resultAPI` is set to `FALSE`. Please change `dump_data_directory` from `/ldas_outgoing/jobs` to `/tmp/ldas_outgoing/jobs`.
4. `wrapperAPI` uses `run_code` to determine some of the directory names under `dump_data_directory`. Change the `run_code` from `LDAS-DEV` to `NORMAL`.
5. The actual directory where files will be written is defined by: `$(dump_data_dir)/$(run_code)_N/$(run_code)` where integer `N = jobID / 10000`; i.e. using default values for the resource variables data for job with `ID=8` would be written to the `/ldas_outgoing/jobs/NORMAL_0/NORMAL8` directory. Create these directories with:

```
mkdir -p /tmp/ldas_outgoing/jobs/NORMAL_0/NORMAL8
```

Create the corresponding directory.

This completes the installation of the `wrapperAPI`.

3 Running your first wrapperAPI job

Now that you have installed the software required to run search code under the wrapperAPI in standalone mode, you want to test everything together. The following steps lead you through your first parallel execution using the wrapperAPI. First, check that you set LAMBHOST environment variable as described above. Once it is set, create a boot schema file for LAM:

```
echo localhost > $LAMBHOST
```

Then, go to the `example` subdirectory of the `lalwrapper` source code and start the LAM daemon:

```
cd $LALPREFIX/src/lalwrapper/example
lamboot -v
```

The output should read something like

```
LAM 6.5.2/MPI 2 C++ - University of Notre Dame

Executing hboot on n0 (naoise.phys.uwm.edu)...
topology done
```

with your computer name replacing `naoise.phys.uwm.edu`. The version of LAM should match that which you have installed in `/ldcg` by following the instructions on the LDAS web pages.

Next, we run the wrapperAPI using the trivial shared object provided in LALwrapper:

```
mpirun -v trivial.schema
```

The verses of *Swinging on a Star* are printed to `stdout`. Congratulations, you have just run your first wrapperAPI job. The next section contains reference information about the file `trivial.schema` and the input file that must be provided to the wrapperAPI even if the data is ignored by the LALwrapper shared object.

Important Note: LAL, LALwrapper and the wrapperAPI are interdependent. Changes to LAL often require changes to LALwrapper if the latter is to successfully compile. Similarly, some changes to the wrapperAPI require changes to LALWrapper. As a general rule, you should always check-out the same version of LALwrapper as you have of LAL. In particular, if you are working with the development snapshot of LAL, you will almost certainly also have to be working with the development snapshot of the other. If the trivial wrapperAPI fails to run, check that LAL and LALWrapper are compatible – a quick review of recent postings to the `lal-discuss` mail archive will help here. If so, and you still have problems, send an e-mail to `mpiteam@gravity.phys.uwm.edu`. We will reply as quickly as possible.

3.1 LAM schema files

There are two files needed by this test. The first is `trivial.schema`, this is a schema file used by LAM to determine how to execute parallel MPI code. The file contains a single line which is not commented. Since this line is too long to display here without breaking it up, we suggest that you open the file with an editor while you read the explanation of the different parts of the line. Here are the arguments and what they refer to:

h LAM argument to specify only run on node where `mpirun` call is made

-np 4 Start up the program (wrapperAPI) as 4 parallel processes. Remember `n0` is the wrapper master and does no search specific processing, `n1` is the search master and `n2–n3` are search slaves.

wrapperAPI The program LAM runs

- mpiAPI="(marfik.ligo.caltech.edu,10000)"** The socket address to connect to the mpiAPI; this is ignored by the standalone wrapperAPI.
- nodelist="(1-3)"** The wrapperAPI only executes the LALwrapper library routines on nodes 1–3.
- dynlib="/home/patrick/lib/lalwrapper/liblداstrivial.so"** The location of the trivial shared object that was installed with the lalwrapper software.
- dataAPI="(datahost,5678)"** The socket address where the datacondAPI is running; this is ignored by the standalone wrapperAPI. Instead data is read in from a file (in ILWD format) which is specified in a later argument.
- resultAPI="(reshost, 9101)"** The socket address to connect to the Event Monitor API; this is ignored in standalone mode and the output is written to a file.
- filterparams="(1,0)"** A comma separated list of parameters needed by the shared object to run the search code.
- realTimeRatio=0.9** The time for execution of the search must be less than 0.9 length of input time series in seconds.
- doLoadBalance=FALSE** Whether or not to do dynamical load balancing. Should always be false when running standalone, since the mpiAPI is required to execute load balancing.
- dataDistributor=W** The WrapperAPI is responsible for distributing the data. Other options are described in Ref. [2].
- jobID=8** A unique job ID; for standalone execution, this must agree with the jobID in the inputFile below. It is handled by the mpiAPI under normal conditions.
- inputFile="wrapper.ilwd"** The location of the file from which data should be read. The data must be in a consistent ILWD format. See the wrapperAPI baseline requirements for examples.

3.2 wrapperAPI input files, e.g. wrapper.ilwd

The input files read from disk must be in ILWD format. (The file for this example is called `wrapper.ilwd`, look at it in an editor.) The trivial example discussed above does not use the data, however the wrapperAPI requires an input file for execution. The structure of these files is described in detail in [2]; please refer to this document should you require additional information.

3.3 wrapperAPI output files

When you run the wrapperAPI in standalone mode as described so far, it will generate one or more output files depending on the LALWrapper shared object it loads:

- `process_n.ilwd` These files contain process information that will be sent to the database when run as part of an LDAS pipeline.
- `output_n.ilwd` These files contain output that was generated by the LALWrapper shared object when it was executed. The number of files and their content will depend on the shared object and the input data.

All other information will be printed to `stderr` and `stdout`; note, however, this mode of output should only be generated when the code is compiled and run with debugging turned on.

Once you want to test code on real data, you will move to running your job on an instantiation of LDAS which can read frame data and pass it through full pipelines. An explanation of how to do that will be provided in the future. (Target date for such a HOWTO is 21 May 2001.)

4 Adding helloworld to LALwrapper

This example explains how to add a shared object called `helloworld` to LALwrapper.

1. If you have just configured and compiled LALwrapper, then you need to clean up before proceeding to the next step

```
cd $LALPREFIX/src/lalwrapper
make cvs-clean
```

2. Create a directory `helloworld` in the `contrib` subdirectory of your `lalwrapper` source distribution:

```
cd $LALPREFIX/src/lalwrapper/contrib
mkdir helloworld
```

3. The `helloworld` directory should contain subdirectories `src`, `include`, `doc`, and `test` and a `Makefile.am` that contains the single line:

```
SUBDIRS = include src test doc
```

Execute the following commands

```
cd helloworld/
mkdir include src test doc
echo "SUBDIRS = include src test doc" > Makefile.am
```

4. Enter the `include` directory and copy `include/Makefile.am` from `contrib/trivial`

```
cd include
cp ../../trivial/include/Makefile.am .
```

Edit this `Makefile.am` and change the line

```
noinst_HEADERS = Trivial.h
```

to list all the header files in your package, for what follows this will read:

```
noinst_HEADERS = HelloWorld.h
```

The header file follows the LAL conventions. An example can be found in Sec. 4.1; add this content to a file called `HelloWorld.h` in this subdirectory.

5. Enter the `src` directory and copy `src/Makefile.am` from `contrib/trivial`

```
cd ../src
cp ../../trivial/src/Makefile.am .
```

In this `Makefile.am` change every `"libldastrivial"` to `"libldashelloworld"`. Change

```
pkglib_LTLIBRARIES = libldastrivial.la
```

to

```
pkglib_LTLIBRARIES = libldashelloworld.la
```

change

```
libldastrivial_la_SOURCES = Trivial.c TrivialSong.h
```

to list your source files, for example

```
libldashelloworld_la_SOURCES = HelloWorld.c
```

and change

```
libldastrivial_la_LIBADD = $(top_builddir)/src/liblalwrapper.la
```

to

```
libldashelloworld_la_LIBADD = $(top_builddir)/src/liblalwrapper.la
```

Because the library is so simple, the single file can contain all four functions. An example can be found in Sec. 4.2; add this content to a file called `HelloWorld.c` in this subdirectory.

6. Enter the test directory and copy `test/Makefile.am` from `contrib/trivial`

```
cd ../test
cp ../../trivial/test/Makefile.am .
```

Change the line

```
noinst_SCRIPTS = TrivialTest.sh
```

to list the names of the test scripts you wish to run, for example

```
noinst_SCRIPTS = HelloWorld.sh
```

These test scripts should use the `happyAPI` to exercise the shared object. Note that the `happyAPI` cannot handle complicated input and output. Only the simplest shared objects can be tested this way. Now, copy `TrivialTest.sh`

```
cp ../../trivial/test/TrivialTest.sh ./HelloWorld.sh
```

and edit the line

```
args=../src/.libs/libldastrivial.so
```

to read

```
args=../src/.libs/libldashelloworld.so
```

Finally, insure that the `HelloWorld.sh` is executable

```
chmod +x HelloWorld.sh
```

7. Enter the doc directory and copy `doc/Makefile.am` from `contrib/trivial`

```
cd ../doc
cp ../../trivial/doc/Makefile.am .
```

Edit this Makefile.am and change the line

```
PACKAGE = trivial
```

to

```
PACKAGE = helloworld
```

The makefile in the doc directory uses the program `laldoc` to extract documentation from the `.c` and `.h` files. There should be two texfiles in the doc subdirectory:

main.tex: this file is used to make the documentation in the contrib/helloworld/doc directory

helloworld.tex: this file is read by main.tex to make the documentation in the contrib/helloworld/doc directory and is also read by `lalwrapper.tex` in the top-level doc directory to make the `helloWorld` documentation in the `lalwrapper.pdf`

The main.tex file is almost the same in all directories. First, copy the `main.tex` from the trivial package

```
cp ../../trivial/doc/main.tex .
```

then edit the file and replace `\include{trivial}` with `\include{helloworld}`.

The helloworld.tex file does three things: (i) it begins a new chapter of documentation for the helloworld package, (ii) it contains general information not extracted from the source or header file, and (iii) it inputs the documentation for each module that is contained in the package.

An example can be found in Sec. 4.3; add this content to a file called `helloworld.tex` in this subdirectory.

Finally, any figures should be included in the doc directory. These figures should begin with the package name, e.g., `helloworldFig1.eps`, `helloworldFig1a.eps`, `helloworldPicture.eps`, etc. Both eps and pdf versions of the figures should be present.

8. Go up to the contrib directory and edit the Makefile.am there

```
cd ../../
vi Makefile.am
```

Add your package to the "SUBDIRS =" line:

```
SUBDIRS = trivial sick load inspiral power helloworld
```

9. Go to the top level and edit the file `configure.in`

```
cd ..
vi configure.in
```

At the bottom of this file is a large section that begins

```
AC_OUTPUT( Makefile \
```

At the end of this section, before the closing brace, add the Makefiles to be generated in your package. To do this, just add the lines

```
contrib/helloworld/Makefile      \
contrib/helloworld/doc/Makefile  \
contrib/helloworld/include/Makefile \
contrib/helloworld/src/Makefile   \
contrib/helloworld/test/Makefile  \
```

10. In `$LALPREFIX/src/lalwrapper` we now recompile `lalwrapper`:

```
cd $LALPREFIX/src/lalwrapper
./00boot
./configure --prefix=$LALPREFIX \
  --with-extra-cppflags="-I$LALPREFIX/include -I/ldcg/include" \
  --with-extra-ldflags="-L$LALPREFIX/lib -L/ldcg/lib"
make
make check
make dvi
make install prefix=$LALPREFIX/stow_pkgs/lalwrapper-new
cd $LALPREFIX/stow_pkgs
stow --delete lalwrapper-howto
stow lalwrapper-new
```

This completes the process of adding and compiling your `helloWorld` library.

11. You can execute this code in the following way. Go to the `contrib/example` directory and make a LAM schema file to run the library:

```
cd $LALPREFIX/src/lalwrapper/example
cp trivial.schema hello.schema
```

This file contains one long line which should not be broken, so beware if you are using an editor which autowraps. Change the occurrence of `libldastrivial.so` to `libldashelloworld.so`, then

```
lamboot -v
mpirun -v hello.schema
```

You should see the words `Hello`, `LSC!` written to `stdout`. Now you have run a wrapperAPI job using a shared object that you have written and added to the `lalwrapper` package yourself. Congratulations!

12. It's now time to start looking at the example search codes in the `contrib` directory. Either `power` or `inspiral` should serve as good examples to get started. Good luck and happy coding.

4.1 HelloWorld.h

```

/***** <lalVerbatim file="HelloWorldHV"> *****/
Author: Brady, P R
$Id: HelloWorldH.tex,v 1.1 2001/04/26 15:24:21 patrick Exp $
*****/ </lalVerbatim> *****/

#ifdef _HELLOWORLD_H
#define _HELLOWORLD_H

#include <lal/LALRCSID.h>
#include <LALWrapperInterface.h>

NRCSID( HELLOWORLDH, "$Id: HelloWorldH.tex,v 1.1 2001/04/26 15:24:21 patrick Exp $" );

/***** <lalErrTable file="HelloWorldHErrTab"> */
#define HELLOWORLDH_ENULL 1
#define HELLOWORLDH_MSGENULL "Null pointer."
/***** </lalErrTable> */

#endif /* _HELLOWORLD_H */

```

4.2 HelloWorld.c

```

/***** <lalVerbatim file="HelloWorldCV"> *****/
Author: Brady, P R
$Id: HelloWorldC.tex,v 1.2 2001/12/06 23:19:00 patrick Exp $
*****/ </lalVerbatim> *****/
#include <stdio.h>
#include <lal/LALStdlib.h>
#include <lal/LALHello.h>
#include <HelloWorld.h>

NRCSID( HELLOWORLDH, "$Id: HelloWorldC.tex,v 1.2 2001/12/06 23:19:00 patrick Exp $" );

// Initialization routine does nothing but checks its arguments
void LALInitSearch(
    LALStatus          *status,
    void               **searchParams,
    LALInitSearchParams *initSearchParams
)
{
    INITSTATUS( status, "LALInitSearch", HELLOWORLDH );
    RETURN( status );
}

// ConditionData routine does nothing but checks its arguments
void
LALConditionData(
    LALStatus          *status,
    LALSearchInput     *inout,
    void               *searchParams,
    LALMPIParams       *mpiParams
)

```

```

{
  INITSTATUS( status, "LALConditionData", HELLOWORLDC );
  mpiParams=NULL;

  RETURN( status );
}

// Apply search is called once on search master and search slaves
// It prints a string to stdout using the LAL function LALHello()
void
LALApplySearch(
  LALStatus          *status,
  LALSearchOutput    *output,
  LALSearchInput     *input,
  LALApplySearchParams *params
)
{
  INITSTATUS( status, "LALApplySearch", HELLOWORLDC );
  ATTATCHSTATUSPTR (status);

  if ( !(output) ){
    ABORT( status, HELLOWORLDDH_ENULL, HELLOWORLDDH_MSGENULL);
  }

  LALHello(status->statusPtr,NULL);
  CHECKSTATUSPTR (status);

  output->numOutput      = 0;
  output->result         = NULL;
  output->fracRemaining  = 0;
  output->notFinished    = 0;

  DETATCHSTATUSPTR (status);
  RETURN( status );
}

// Finalize routine does nothing but checks its arguments
void
LALFinalizeSearch(
  LALStatus          *status,
  void               **searchParams
)
{
  INITSTATUS( status, "LALFinalizeSearch", HELLOWORLDC );
  RETURN( status );
}

```

4.3 helloworld.tex

```

\chapter{Package: \texttt{helloWorld}}
This shared object prints does very little.
\newpage
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\section{Header \texttt{HelloWorld.h}}

```

```

\label{s:HelloWorld.h}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\noindent Provides routines to be executed by the wrapperAPI which prints
"Hello, LSC!" to stdout.

\subsection*{Synopsis}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\subsection*{Error Conditions}
\input{HelloWorldHErrTab}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\subsection*{Structures}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\subsubsection*{struct \texttt{MyStruct}}
\index{\texttt{MyStruct}}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\noindent Stuff about the structure .....

\begin{description}
\item[\texttt{TYPE element}] What it is .....
\end{description}

\vfill{\footnotesize\input{HelloWorldHV}}
\newpage
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\subsection{Module \texttt{HelloWorld.c}}
\label{ss:TimeSeriesToTFPlane.c}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Contains the four LAL functions neede to run search code .....

\subsubsection*{Prototypes}
\vspace{0.1in}
Autodoc your prototypes if there are any ..... see the LAL spec for how to do
this.

\subsubsection*{Description}

\subsubsection*{Uses}

\subsubsection*{Notes}

\vfill{\footnotesize\input{HelloWorldCV}}

```

References

- [1] MPI working group, *LAL-LDAS Interface Coding Specification*, distributed as part of `lalwrapper` package and LIGO-T010003-00.
- [2] Kent Blackburn *et al*, *The wrapper API baseline requirements and implementation*, LIGO-T990097-13.
- [3] Bruce Allen *et al*, *Numerical Algorithms Library Specification and Style Guide*, distributed as part of `lal` package and LIGO-T990030.