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

Document Type LIGO-T970100-A - E

LIGO Data Analysis Software Specification Issues

James Kent Blackburn

Distribution of this draft:

LIGO Project

This is an internal working document
of the LIGO Project.

California Institute of Technology
LIGO Project - MS 51-33
Pasadena CA 91125
Phone (818) 395-2129
Fax (818) 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/

1.0 Introduction

The subject material discussed in this document is intended to support the design and eventual implementation of the LIGO data analysis system. The document focuses on raising awareness of the issues that go into the design of a software specification. As LIGO construction approaches completion and the interferometers become operational, the data analysis system must also approach completion and become operational. Through the integration of the detector and the data analysis system, LIGO evolves from being an operational instrument to being an operational gravitational wave detector. All aspects of software design which are critical to the specifications for the LIGO data analysis are introduced with the preface that they will be utilized in requirements for the LIGO data analysis system which drive the milestones necessary to achieve this goal.

Much of the LIGO data analysis system will intersect with the LIGO diagnostics system, especially in the areas of software specifications. Both the data analysis and diagnostics are expected to evolve; the diagnostics as a result of increased insight into the workings of the interferometers, and the data analysis as a result of the increasing understanding the astrophysical models that produce detectable gravitational waves. For these reasons, and the efficiency of having a common set of software specifications for the LIGO Project, the specification outlines in this document apply with nearly equal weighting to the LIGO diagnostics system and to the LIGO data analysis system.

2.0 Scope

The scope of this document is to present the issues which will drive the specification for the LIGO data analysis software with the emphasis being on software. It is the intent of the discussions that follows to give support to resulting specifications on the bases of current design and development tools available to the data analysis software. It is not the scope of this document to specify the design of the data analysis system nor to address the particular components with sufficient detail to warrant this as a specification or requirements document.

A small set of assumptions are presented in the next section to give the scope of the challenge facing the LIGO Data Analysis System and the software developed as an integral part of that system. This is not intended to be a complete list. Other inputs may become important to the statement of software issues in the future. Some of the assumptions below may become insignificant due to unforeseen changes in how the data is collected or as a result of advances in technologies for writing and reading data.

3.0 Data Analysis Assumptions

The bulk of LIGO data will be collected from a multitude of channels consisting of lists of 16 bit values sampled at data rates up to 16384 (*possibly 32768 in the future*) samples per second representing the physical behavior of the interferometers as a function of time. Additional information about the state of the interferometers will be recorded in conjunction with the time series data channels, including the calibrations associated with filtered channels and instrument state information.

Fundamental assumptions that influence the data analysis software specifications are

1. Data collection rates from each interferometer will approach 6 megabytes per second. (*Increasing the sample rate to 32768 on the fast channels results in ~ 1% increase in data.*)
2. Of the total data collected, less than 10% will have immediate relevance to the detection and measurement of gravitational waves.
3. All data will be collected at sample rates corresponding to a “power of 2” samples per second. The importance in this being that FFT performance is substantially enhanced.
4. Data will be delivered in the form of frames; Each frame will completely (possibly through indirect reference) represent the information needed to analysis the scientific and diagnostic characteristics of the interferometers.
5. No particular requirements will be imposed through this document that distinguish between “on-line”, “nearly on-line” and “off-line” data analysis as a part of this specification. The differences between these will be considered a function of resources and not specifications on the software.
6. Compute requirements which influence LIGO software standards will be general in scope allowing both base-line methods as well as novel strategies, hence providing for both production and research software.

4.0 Data Format Specifications

LIGO has adopted a data format characterized by two overlying features. One is that the data will be grouped in time into structures known as frames. Each frame file is a complete record of the informational content of the experiment, including all physics and all instrumental signals. The frame will be of a fixed length consisting of a “power of 2” seconds in length to maintain a “power of 2” number of samples for each time series signal being captured. The advantage in this being that the discrete representation of this time series data in the frequency domain will be carried out using fast Fourier transform (FFT) methods which are superior in performance to other methods for arriving at the frequency representation of discrete time series data. The second overlying characterization of the data format is that the data stored in each frame have a hierarchical structure which will reflect the organization of the interferometer with all of its subsystems. This will allow for a more efficient interface into the units of data stored within each frame and facilitate the development of important diagnostic and veto “super” signals to characterize detector performance.

The implementation of this hierarchical structure will be based on the structure data type from the C language. The use of the native C structure makes for a very efficient and fast translation of data found on a mass storage media, whether it be random access hard disk or sequential tape, into and out of computer memory. A common data format I/O library for writing and reading this format is currently being developed by the VIRGO project in collaboration with LIGO. This will allow data analysis methods developed by both programs to be able to input from either projects data with minimal (*channel naming conventions, seconds per frame, etc.*) or possibly no modification.

In the event that the two projects diverge in the definition of the C-structures or version of the I/O library at some future date, the adoption of the C structure for the hierarchical organization of data will standardize the data formats sufficiently to make simple translations possible. Additionally, the distribution of the data to scientist and researchers in the community at a future date may be made more practical by changes in computers, networks, communications and even the World Wide Web in another to be determined format. Again, the fundamental standardization on the C structure will provide a more natural point of departure into distribution format.

5.0 Platform Specifications

LIGO data analysis software should not be so targeted towards a particular platform as to become totally dependent on that platform. This restriction can be relaxed greatly when a sufficiently high level of standardization and modularity are incorporated into the design of the system. In addition, current computer architectures support a broad mix of operating systems and hardware, for example, Solaris is available on Sun SPARCstations as well as Intel based PCs. This de-couples much of the dependences between operating systems and hardware that have been and to a lesser extent still exist on some platforms.

5.1 Operating System Specifications

The popularity and longevity of Unix make it a very attractive choice for the LIGO data analysis system. For this reason and others related to hardware, the Unix operating system will be the targeted environment for LIGO data analysis software development and implementation.

There are many “flavors” of unix available on a wide range of hardware platforms. Because of this, it is important to either restrict the software to the subset of functions that are common to all Unix systems or restrict the type of Unix to those that are following an accepted standard. Obviously the second of these is the more attractive. Of the many flavors of Unix, the AT&T System V variant has become the most popular and mature. The interaction of software with the operating system has received a great deal of attention in the last decade. The Portable Operating System Independent Interface (POSIX) standard was developed to insure portability of software which utilized functionality through the operating system. By standardizing all software to use POSIX library calls to the operating system, LIGO data analysis will be guaranteed a minimal effort in porting software between Unix operating systems. There are varying levels of POSIX compliancy that is found in operating systems and LIGO must select a level that is most up to date. As of the fall of 1996, Solaris 2.5 was fully compliant. This should be fully investigated when considering other versions of Unix.

5.2 Hardware Specifications

LIGO data analysis is substantial, requiring on the order of several hundred of gigaflops of performance per interferometer output to cover the range of stellar masses in the limits of current theory (>0.2 solar mass stars) in the case of binary inspiral searches. Pulsar searches will be much more demanding. The storage of LIGO data also requires substantial resources if the full six megabytes per second of data collected at each interferometer output is to be recorded for extended periods of time. Hardware considerations should be left fairly general until the actual time arrives to make the purchase, using systematic benchmarks during development to test algorithms and portability. This is because the performance curve associated with expense is such a steeply rising relationship that dedication to a particular hardware platform at this time would be unwise.

There are several hardware options that should be considered in the development of data analysis software which will begin long before the purchase of the actual hardware to be utilized by LIGO in the data analysis system. The most significant of these is that the software be capable of running on some form of distributed computing architecture. This could involve massively parallel computers or a network of distributed workstations. Cost and inter-node communication rates will play an important role determining if one architecture has a clear advantage over the other at the time that hardware is procured. In the meantime, very flexible solutions exist which allow many distributed computational problems to be developed for both distributive networks of workstations and on massively parallel computers. One of the most popular solutions at present is the Message Passing Interface (MPI) of which more will be said in the next section.

Another hardware consideration that always comes up when low level optimization become important or when binary data types are being used is the issue of “big-endian” and “little-endian”. These are choices made in hardware which address which byte in a word is to be defined as the most significant. Most of the RISC hardware Unix originated on uses the “big-endian” choice. However, there are two hardware contenders at this time that are based on the “little-endian” choice. These are the DEC ALPHA and the Intel x86 processors. The LIGO data analysis system can not assume that the hardware will be one choice or the other in the design and development of low level routines. This is particularly true of the software associated with reading and writing the LIGO frames.

6.0 Interface Specifications

LIGO data analysis software can be divided up into various categories based on functionality. There will be components that handle the interface with the users, interface with the recorded data from the interferometers, perform the calculations necessary to search for gravitational waves, interface with the hardware and operating system and display/play the results to name several.

Many of these components will be optimized in some way to make the data analysis system as efficient and cost effective as possible. This requires that the function of each component be well defined and that the data types and messages between the components have a strictly followed interface specification.

A basic constraint that is assumed in the interface flowchart is that the LIGO Data Analysis System will see a data flow that is one way, as illustrated in Figure 1. This prevents the Data Analysis System from interfacing back onto the Data Acquisition System, a constraint that is not in place for the LIGO Diagnostic System which will need to inject and stimulate the interferometers and their subsystems to complete the diagnostic cycle through a tightly managed software interface.

.

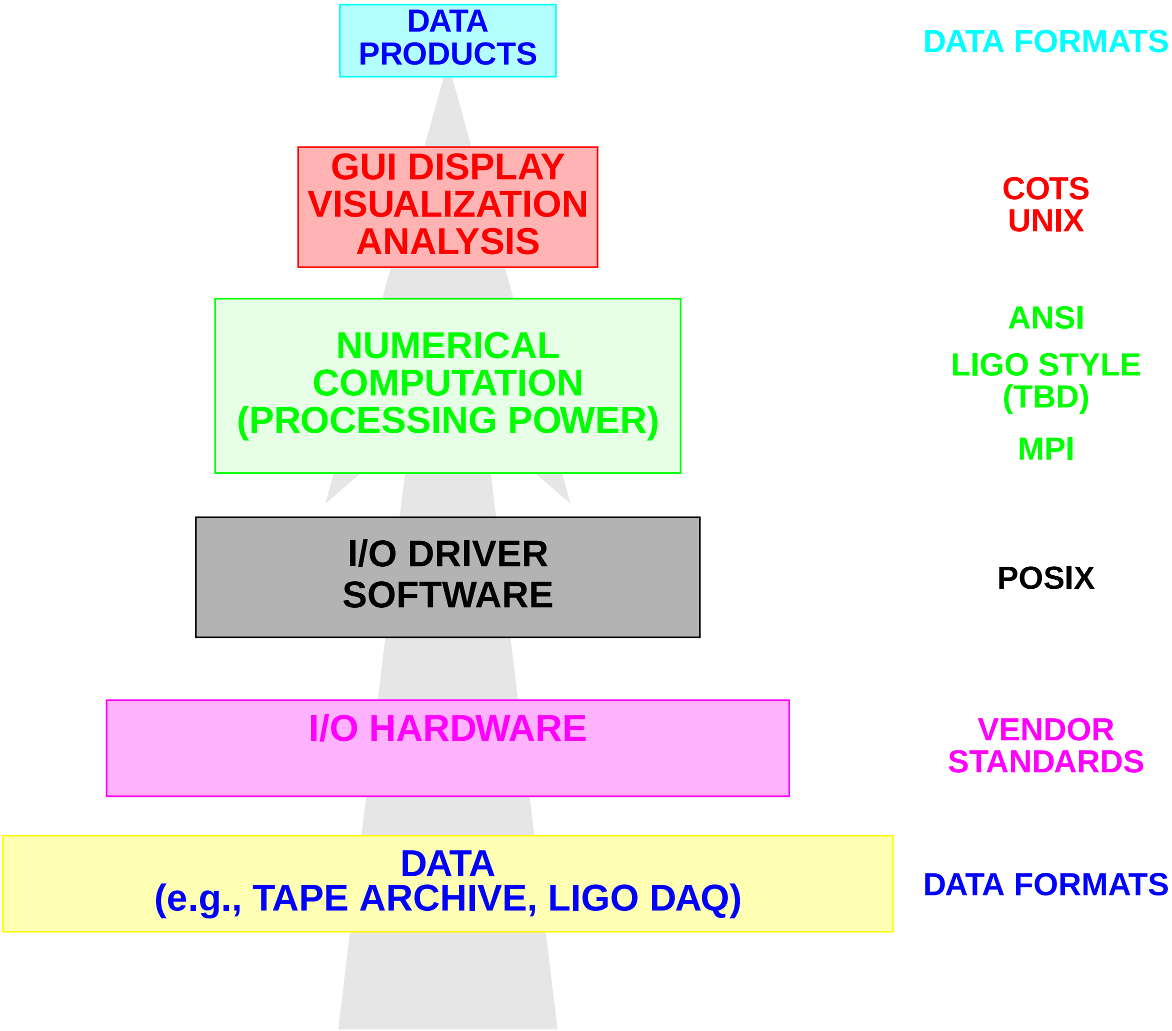

Figure 1: LIGO DAS Software Interface Flowchart

6.1 Modularity

Another important aspect of the data analysis system is modularity. This applies not only to the categories or components that the system is divided up into, but also the individual elements or functions available within each category. The most successful paradigm for achieving modularity is the use of objects and object oriented programing. Central to the ideas of object oriented programing are the encapsulation of data, protecting the data from arbitrary functions, and the integration through class methods of the functions to be performed on the data with the data. Communications between instances of classes is handled by message passing. Objects are designed with the necessary modularity to provide the necessary functionality and data content for modularity. The most popular language for developing and maintaining modular, object oriented software is C++. However, through excellent coding standards and practices (style), it is possible to implement many of the beneficial aspects of modularity in C or FORTRAN code. Key to being able to accomplish this is to define a fixed set of data types to be communicated between modules and design a module interface around these data types. This should exclude the use of any global variables for passing information and avoid the use of macro values. The organization of the parameter arguments in procedural calls must also be well defined to guarantee modularity.

6.2 Libraries

Libraries can be thought of as a collection of procedures or modules that are tied together as an integrated unit. The design of libraries is critical to their successful use. LIGO should avoid developing a large number of libraries for data analysis. Several operating systems, including Unix link in libraries dependencies in a single pass. This means that the ordering of multiple libraries is important. To avoid this these rules should be observed:

- do not develop a multitude of libraries,
- do not develop libraries that have routines that depend on routines that exist in other libraries,
- if absolutely necessary, designate a library that will always be parsed last as the library to contain inter-library dependent routines.

These rules will minimize the confusion that using libraries can generate. However, it is absolutely mandatory that all routines within a library be completely documented along with all dependencies.

A second area for trouble with using libraries under Unix is in the size of the executable image. Individual modules or routines found in a library are loaded into the executable in units of size such that all routines compiled together in a common file will be loaded together. It is an unwise approach to collect all subroutines and functions to be used in building a library into a single file. This would result in every module being loaded into the executable at link time whether it is needed or not giving a excessively large final executable size. To avoid this all modules to be archived in a library should have separate files unless they are guaranteed to always be used together. It is possible to use or develop tools which can split out each individual routine found in a source file and then separately link them and archive them into the library so that only the individual required modules are loaded into the executable, but care must be taken when working with a large number of modules not to overflow the cache buffers that store the list of extracted modules from the source or the library will be incomplete after the build procedure.

6.2.1 Module Libraries

The Unix operating system is supplemented by an enormous, ever-expanding base of modular utilities (*programs*) that users manipulate through scripts, redirection and pipes to build increasingly complex computational units. This modularity is guaranteed through the common treatment of standard input and standard output as a stream of data that is parsed, interpreted, processed and reconstructed as a stream for output based on a ‘first in first out’ (FIFO) data flow. No special meaning exists for the stream prior to or subsequent to the processing by the modular utility. Through this paradigm, Unix utilities exist as a module library. What this paradigm offers that the conventional concept lacks is that each routine within the module library is fully functional entity. Having data communicated between modules through a stream based on a FIFO is somewhat restrictive and prevents randomly selecting or accessing data within the stream without buffering large segments, however, much of the LIGO data is generated in the form of time series which are inherently FIFO in origin so this restriction should not be a major detriment to many if not most aspects of the LIGO data analysis system. The use of module libraries of this form will be very practical to the LIGO data analysis system and easily implemented on top of the underlying Unix operating system.

6.2.2 Numerical Libraries

Many of the modules or routines to be found in LIGO data analysis libraries will be numerical in nature and compute intensive. To better optimize the performance of these libraries, they may be developed in specific languages which must be accessed by calls from data flow software developed in a different language. Passing data between routines compiled in different languages depends upon platform specific details and in general will not be portable. Making the library useful on other platforms will be facilitated with a few minor restrictions on the argument lists being shared between the calling routine and the library routines:

- data types passed should be simple universal types such as integers, floats, doubles, and one dimensional arrays of these,
- avoid passing strings; FORTRAN treats a string closer to a C structure than an array of characters,
- do not pass composite data types, i.e. use the common *simple* data types like int, integer, etc..

It should be a easy task to design the routines found in numerical libraries based on these rules without loss of functionality. This will allow choice of the language used to implement the numerical algorithm to be made optimally.

6.2.3 Class Libraries

Class libraries are a collection of object oriented modules. The basics of the object oriented paradigm (OOP) is the organized containment of data and the procedures which operate on the data (called methods or member functions in the paradigm) into objects which protect the data from tampering by procedures other than the ones specified by the class. This is a very powerful paradigm affording many highly practical side effects. In order for OOP to be successful many powerful features are integrated into the object approach which go a long way towards increasing the reusability of code and the maintainability of large software systems. LIGO should take OOP languages very seriously as a candidate for developing the skeletal system which is responsible for overseeing and integrating the whole of the system. In such a context many of the fundamental data types which are globally significant to the data analysis system should be developed as objects which are instantiated from classes bundled in the LIGO class library. These data types might be time series, spectra, arrays and other composite data types.

Developing a class library in an object oriented language such as C++, Objective C, or small talk is by no means necessary. However, it does add an additional level of syntax and guarantee data encapsulation that must be provided by extra software and devotion on the part of the programmer in non object oriented languages. **For this reason, it is highly recommended that LIGO adopt C++ for its maturity in any components that will be considered class libraries.**

6.2.4 Dynamic Linked or Shared Libraries

Most modern operating systems, including Unix support dynamic linking, a method for invoking library modules at run-time. (A library module is a binary file that contains the executable library routines.) A library module that can be loaded in this manner is called a “dynamic-link library,” or DLL. Advantages of dynamic linking include

- the promotion of code reuse through logical and physical modularization,
- the reduction in the amount of hard disk space that is used because a single DLL can be shared among multiple processes,
- the reduction in the amount of RAM that is used because a single DLL can be loaded once into physical memory and shared among several concurrent processes,
- they allow modular, seamless upgrading of a program because one or more DLLs can be replaced with newer versions. You do not have to rebuild an entire application,
- they greatly benefit localization by placing language-specific information into DLLs, an application can simply load the appropriate language DLL to retrieve information.

It is important to note that without careful management, DLLs can cause version-control and configuration problems. DLLs must have an associated version number and software must be certified to work properly with specific versions of DLLs. This will be discussed in greater detail in the section on Configuration Management to follow.

6.3 Application Programming Interfaces

The application programming interfaces (API) are a set of routines that an application uses to request and carry out lower level services performed by a computer’s operating system. Many of these services under Unix are standardized by POSIX. For computers running a graphical user interface (GUI), the API manages the windows, icons, menus and dialog boxes. LIGO will undoubtedly desire an operating system supporting a graphical user interface. The standardization of the API must provide portability across various hardware vendors and be complete and flexible enough to carry out the lower level services that are needed by LIGO data analysis software. Many commercial API products are available. User interface APIs such as X Windows and Motif are also standards for Unix operating systems. The final decision on the particular GUI API has not been made, though Motif is emerging as a standard at this time in the Unix environment. No matter what choice of API, the software must be developed with a clear understanding that code that uses an API be layered independent of other codes that handle such functions as numerical computations and analysis.

6.4 Commercial Analysis Packages

Commercial analysis and data visualization packages will play an important role by providing user flexibility and interaction with data and data analysis. The Data Analysis system will support those packages which provide APIs and libraries for standard ANSI languages such as C and FORTRAN. This is a very common feature found in Mathematica, Matlab, AVS, IDL and many other commercially available packages. In addition the Data Analysis system will produce data products which can easily be imported or translated into a native commercial data format.

6.5 Distributed Computing

Many aspects of LIGO data analysis, such as the binary inspiral template search have an inherent parallelism. This makes this type of analysis ideal for distributed computing, whether by a network of workstations or by specialized massively parallel hardware. There are several routes to take in designing software to work in a distributive environment. Distributed computing has been a part of Unix since its earliest days. Many routine activities such as email, ftp, telnet, etc. rely on computers sharing resources at some level. This is managed at the lower levels by Remote Procedure Call (RPC). More sophisticated and structured solutions for distributed computing exist, each with advantages and disadvantages. An emerging standard which is emerging in the supercomputing community is the Message Passing Interface (MPI). MPI allows distributed computing code to be developed in C++, C and FORTRAN based on a library of routines for passing data (messages) back and forth between processes running on different processors (nodes). The library is developed to work on massively parallel computers as well as networks of workstations. Once the code is developed under this paradigm, it is a fairly simple if not trivial task to port the code. MPI is sophisticated enough to allow heterogeneous nodes to work in parallel and the number of nodes is only limited by the number of workstations you can arrange to access. Implementations of MPI are available from operating system vendors, as well as from university programs around the world. A useful reference to MPI implementations on the web is **http://www.osc.edu/mpi**. The current version of MPI is 1.1, and MPI 2 will become available at some point in 1997. Most implementations support the full set of functions under this version.

An older form of distributed computing is based on Parallel Virtual Machines (PVM). MPI is emerging as the standard for distributed computing on massively parallel machines and is also available for clusters of workstations. It has better performance in general than PVM, though it is less flexible. **MPI should make a better long term environment for distributed computing.**

More recently, as a result of the popularity and growth of the World Wide Web (WWW), JAVA has become a popular environment for doing distributive computing. Any computer with a web browser is a potential node throughout the whole of the internet. JAVA also bundles a true object oriented language with a graphical user interface. JAVA code can be run as stand-alone or as an applet controlled through a web browser. JAVA performance is somewhat limited being that it is interpreted, however, the code is highly portable, not only between traditional Unix platforms, but also among the PC Windows platforms.

The Open Software Foundation (OSF) is commercially developing the Distributive Computing Environment (DCE) based on the server/client model. It also aims at being operating system independent, allowing heterogeneous environments. DCE has flexible security mechanisms integrated into the model based on Kerberos version 5 which have made it attractive in applications being developed at Los Alamos. Its applications do seem to be limited to clusters of workstations at this time.

Distributed Objects is a new paradigm for distributed computing which combines object oriented with server/client technologies. This paradigm allows objects to be distributed across heterogeneous networks of computers. Objects traditionally interact with one and other through messages. When the Objects are distributed over the network, the network becomes the computer in this distributed computing paradigm. The inherent advantages of distributed objects is that it provides a natural mechanism for distributed computing with object oriented languages. However, many of the distributed object environments are commercial and tightly coupled to a particular operating system, though in general not tightly coupled to a particular hardware platform.

6.6 Graphical User Interfaces

Most of the effort involved in designing, implementing and maintaining an integrated software system is consumed by the graphical user interfaces (GUI). Though the percentage of effort has decreased greatly in the last few years, the effort is still substantial. A part of this is the need to support customizing of the graphical environment tailored to the immediate needs of the user.

To as great an extent as possible, the graphical user interface should be based on an already available package. The package should be current, well documented and maintained. It should provide an interface to user specified code, making the GUI a modular, layered component of the LIGO data analysis system which can easily be replaced by a comparable package at the discretion of the LIGO Project or end users of the data analysis system that are willing to develop a new generation GUI.

Several aspects of the functionality of the GUI should be standardized. The used of various widgets such as pointers, buttons, sliders, dials, etc. should be uniform and intuitive. Rules for color coding and color contrast should be established with the intent of maximizing information in a uniform manner. Examples include the standardization of colors like green, yellow and red to warn of the state of a component in the data analysis system.

7.0 Standardization

Through a good choice of standards, software portability, maintainability and longevity are assured. Beginning in the late 1980s, several standards for UNIX-like operating systems and the C language were proposed. Most of the standards define an operating system environment for applications based on C or C-like languages. They do not address applications based on other languages such as FORTRAN. These standards do not impose dramatic changes in venders' systems and developers' software, instead they provide a common backdrop for the industry. The two most significant of these are the ANSI C standard from the American National Standards Institute and POSIX from the Institute of Electrical and Electronic Engineers (IEEE). These have become sufficiently widespread in use that most computer venders today provide UNIX systems that conform to the ANSI C standard and to the POSIX standard (at least at the subset level of POSIX.1). Because of this conformity to standards by the major computer vendors, it is extremely important that LIGO adopt these as LIGO standards. It is important to understand that if done early, these standard are not in themselves sufficient to guarantee software portability and maintainability, but they will make a significant contribution to achieving that target. It is also important to understand that these standards are not a burden to adopt and will not limit the functionality of software developed under them. As a final comment about these two standards (which will be discussed in more detail later), they do not address issues of graphical user interfaces (GUI). The leading standards for GUIs, such as TCL/TK and Motif, in the UNIX world are based on X11. Several APIs exist that are based on X11. The most popular of these at this time appears to be Motif, which provides a standard "look and feel" to the windows and widgets that make up a GUI. Given that LIGO has adopt a UNIX operating system, the most likely GUI standard will probably be based on Motif and the Motif widget set whether it is developed in house or provided by a 'Commercial

Off The Shelf" (COTS) product. However, OpenStep is rapidly emerging as a competitive environment support both Unix, Macintosh and Windows and should be reviewed.

7.1 Non-Compliance with Standards

It is possible, though it should not be likely, that several of the standards being discussed introduce excessive hardships on the design of the Data Analysis System. If such a standard is found, the decision to allow non-standardization within the system should be approved by a panel or change board representing the interests of LIGO and the Data Analysis Software Standards. If the panel approves the use of the non-standard feature, then it must be documented. A document containing the approved exceptions must be maintained and distributed.

7.2 ANSI C Language Standard

In 1989, the American National Standards Institute proposed the C programming language standard X3.159-1989 (ANSI C) to standardize C language constructs (headers and structures) and libraries. The previous implementations of the C language supported computer venders was based on the constructs and libraries proposed by Brian Kernighan and Dennis Ritchie (K&R C) which is still available for many UNIX operating systems. The major differences between K&R C and ANSI C are that ANSI C implements:

- function prototyping,
- support of const and volatile data type qualifiers,
- support for wide (16 bit) characters and internationalization,
- permits function pointers to be used without dereferencing.

These improvements over K&R C are significant not only because they are a part of the standard, but also because they improve the quality of software and reduce some of the debugging associated with the more flexible K&R C.

The ANSI C language standard is also of importance to the C++ language. This is because the standard class libraries of C++ do not provide the functions found in the standard C libraries (e.g., get time of day, or the use of the strlen function, etc.). These functions are derived from the ANSI C libraries without complication by C++. Additionally many of the ANSI C language standards were adopted from the proposed standards for C++, demonstrating their common origins as being bi-directional.

7.2.1 ANSI/ISO C++ Standard

In the early 1980s, Bjarne Stroustrup of AT&T Bell Laboratories developed the C++ programming language. It was derived from C and incorporated object oriented constructs such as classes, derived classes (inheritance), and virtual functions from simula67.

Since then C++ has gained wide acceptance in the software industry, becoming a major software development tool even before a standard was drafted. It wasn't until 1989 that the X3J16 ANSI committee drafted an ANSI C++ Standard. The International Standard Organization (ISO) WG21 committee joined with the ANSI X3J16 committee to publish a draft version of the ANSI/ISO C++ Standard in 1994. This C++ standard in still in the development stage but should become an official standard in the very near future. As it stands today, most commercial C++ compilers are based on the AT&T C++ language version 3.0 or later and are compliant with the draft ANSI/ISO

C++ standard. As such, these compilers should support C++ classes, derived classes, virtual functions, operator overloading, template classes, template functions, exception handling and iostream library classes. It is important to note that the C++ standard defines the binding with the ANSI C language providing full interoperability between the two languages. This is the only case where this level of binding specification exists.

The C++ language does not in itself guarantee object oriented software development. This is primarily because C++ encapsulates C with object oriented features. To achieve the benefits of object oriented design, great care and discipline must be exercised in the code development cycle. This equates to a large start-up investment and strict coding rules. Object oriented code development must proceed as follows:

1. design the objects first
2. design the hierarchy next (bottoms up strategy)
3. design the capabilities of the base classes
4. design the data structures last
5. develop the code

Because of the necessity and overhead of a good design, object oriented projects come with a large initial phase overhead. It is not unusual to see months of design effort before a single line of code is written. However, the slow start-up is more than made up in the later phases of the software development by having a truly object oriented design. Maintainability, reliability, extensibility and portability in a mature project require minimal effort as indicated by the trends shown in the figure below comparing object oriented design(C++), structured design(C) and procedural design(Fortran) illustrating the cross-over where object oriented design begins to pay off.

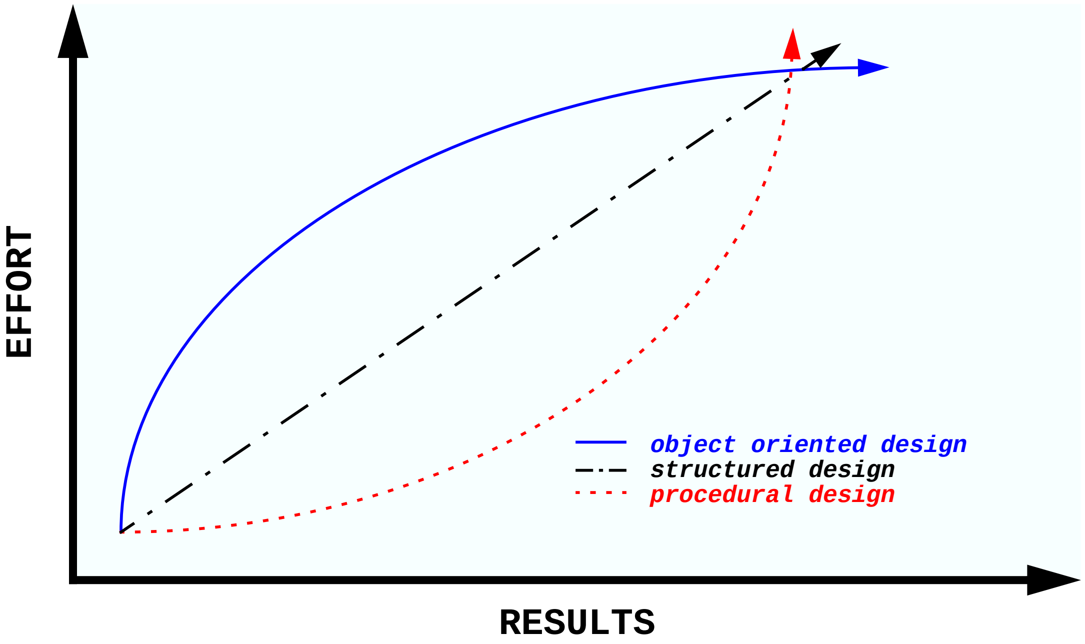

Figure 2: Comparison of code development for various software designs

7.3 Fortran Language Standard

The history of Fortran is ancient in comparison with C and C++. This longevity of the language is most impressive and signifies the importance it has had in the areas of computational. Fortran was originally developed by John Backus of IBM in the 1950s as one of the earliest high level computer languages. The widespread use of Fortran lead to many dialects evolving that caused difficulties with the exchange of Fortran programs. This lead to the ANSI committee X3J3 forming in 1966 the Standard which is now known as Fortran 66. This same committee formed the Fortran 77 Standard in the 1978. Both the Fortran 66 and Fortran 77 languages were poorly implemented by many vendors and in themselves contained many flaws which led to the development of extensions to the language and the development of new Fortran-like languages such as RATFOR which used preprocessors to produce Fortran code that could be compiled with a standard Fortran compiler. The change-over from one Fortran standard has always been slow and pressure always existed for vendors to support more than one standard at a time. This has not changed with the recent publication of Fortran 90 by the ANSI/ISO which was originally to be called Fortran 80.

Fortran's superiority has always been in the areas of scientific and engineering computing. But after nearly thirty years of existence, it is no longer the only computer language being used by scientist and engineers. Computer science has evolved greatly in this span of time. There was a need to integrate some of the new ideas from other languages into Fortran. As a result the standardization of Fortran 90 was much more of a development process than an effort to standardize existing practices in the language. This lead to a more flexible source format, ease in working with arrays, standardization of Fortran 77 extensions used by vendors and the adoption of useful components of other languages such as pointers and user derived data types. The joint ANSI/ISO committees, after much dissension and many failed attempts to adopt a standard, announced the formal publications ISO/IEC 1539 of July of 1991 and ANSI X3.198-1992 of what is now known as Fortran 90.

Having experienced the hardships associated with redeveloping Fortran to the Fortran 90 standard, the committees have adopted a new mode of operation requiring firm dates for the release of new standards and a policy that dictates that any planned functionality be dropped from a new standard if it does not demonstrate sufficient maturity instead of the previous policy of debating and waiting. This has lead to a minor revision by the committee of the newest Fortran 95 standard.

At this time the ISO committee has decided that these three features of Fortran,

- handling floating point exceptions,
- inter-operability with C, (*critical to LIGO in my opinion*),
- permitting allocatable arrays as structure components, dummy arguments, and function results,

are so urgently needed that it has established a development body to address the technical issues. These technical issues will be integrated in the next release of Fortran planned for the year 2001.

Of these, the need for inter-operability with C is the most limiting aspect of Fortran for LIGO. Binding modules of code developed in both Fortran and C is not standardized across vendors of computer languages. At present, a complex C header package known as "cfortran.h" is available from CERN to handle the differences in calling routines between the two languages on many of the popular platforms of the late 1980s and early 1990s. However, the environment provided by cfortran.h is not updated frequently enough and is not bug-free enough to justify adopting it as a standard for binding C and Fortran. Until the ANSI committee establishes standards for interoperability with C, each Fortran module to be implemented in the LIGO Data Analysis System will have to be carefully “wrapped” and verified by programming staff to assure a clean, modular interface between the two languages. In addition, the wrapper should be placed around the Fortran module, making it look like a C module to make it a supported module to C++, as well as C.

It is also worth pointing out that functionality and features have been deleted by the committee from previous standards of Fortran. This type of standard evolution leads to translation of existing code, or even less attractive, a decision to stay with a previous unsupported version of Fortran.

In summary, it is clear that there are inherent risks to be found in even the highly standardized languages like Fortran which should be weighted proportionally when considering the risks of a language standard in draft form as is the case with C++. **For these reasons only languages which provide full inter-operability with C should be supported by the data analysis software.**

7.4 POSIX: Operating System APIs

There are many versions of UNIX in existence today. Each one provides its own set of application programming interface (API) functions. This makes it difficult to develop applications that will port easily between the different UNIX versions. To overcome this difficulty, the Institute of Electrical and Electronic Engineers (IEEE) formed a special task committee called POSIX (Portable Operating System Interface) in the 1980s to create a set of standards for operating system interface functions. The committee consists of several subgroups such as POSIX.1, POSIX.1b and POSIX.1c. Each addressed the development of different aspects of operating system interfaces. The division is as follows:

- POSIX.1 subgroup proposes standards for base operating system application programming interfaces, e.g., APIs for file and process manipulations. It is formally known as IEEE standard 1003.1-1990 and has also been adopted by the ISO as the international standard ISO/IEC 9945:1:1990.
- POSIX.1b subgroup proposes standards for real-time operating system interfaces, e.g., interprocess communications. It is formally known as IEEE standard 1003.4-1993.
- POSIX.1c subgroup proposes standard for multi-threaded programming interfaces. This is the newest POSIX standard and provides for concurrent processing of segments of code.

Much of the focus of the POSIX committee is based on UNIX, but the standards that they propose are for generic operating systems. For example, VMS, OS/2 and Windows NT are not UNIX operating systems, however they are POSIX compliant. Most current UNIX operating systems, including Solaris 2.4 which is the current LIGO development platform, are POSIX.1- compliant. This does not prevent the UNIX vendor from providing their own APIs which are specific to their systems.

Because of this it is necessary to restrict the operating system API functions used in development to the subset of a vendors APIs which are POSIX compliant to guarantee portability. This is not as daunting a chore as it may sound. To ensure that a user program conforms to the POSIX.1 standard, the user should either define the MACRO constant _POSIX_SOURCE at the beginning of each source (C or C++) module of the program prior to the inclusion of any headers as:

```
#define _POSIX_SOURCE
```

or specify the -D_POSIX_SOURCE option to the C or C++ compiler in a compilation as in:

```
% CC -D_POSIX_SOURCE -o prog *.cpp -lm
```

Either of these will provide a constant that is used by the preprocessor to filter out ***all*** non-POSIX.1 and non-ANSI C standard codes from headers used by the program to define the API functions used to interface with the operating system.

In addition, many C and C++ compiler vendors provide an option, usually `-posix`, that can be used to achieve the same filtering (check your compiler manpages).

POSIX.1 and POSIX.1b define system parameter limits in the form of manifested constants. These can be found in the <limits.h> header file. The purpose of these constants is to set lower limits on various system configurations. The POSIX standards specify that these lower limits shall not be less than some value, but do not restrict them from being larger. In this way, an application is guaranteed to have a minimum level of resources on all systems. Table 1 below presents a list of these that are most likely to be used by the LIGO Data Analysis System. A complete list can be found in the POSIX standard.

Table 1: Probable POSIX.1 Constants in <limits.h> header effecting LIGO DAS

POSIX.1 Manifest Constant	Min Value	Meaning
_POSIX_CHILD_MAX	6	Max number of child processes created by one process
_POSIX_OPEN_MAX	16	Max number of files opened by one process
_POSIX_STREAM_MAX	8	Max number of I/O streams opened by one process
_POSIX_PATH_MAX	255	Max number of characters in a path name
_POSIX_NAME_MAX	14	Max number of characters in a file name
_POSIX_PIPE_MAX	512	Max size in bytes of data used in pipe file for a single read/write

Others may be important, but these tend to affect the most common interface functions used. The level of portability that is provided by POSIX makes it a necessary component in the design of any heterogeneous software system. It is highly advantageous to adopt this standard and should be a key component of the overall LIGO Data Analysis System.

7.5 Motif Toolkit/Environment

Motif is a proposed standard for user interfaces (GUIs) developed by the Open Software Foundation (OSF), a non-profit organization of companies whose charter calls for the development of technologies which will enhance inter-operability between computers from different hardware vendors.

Motif consists of a programming toolkit that is a collection of pre-written functions that implement all the features and functions of a particular instance of a GUI. It is based on X Windows and the X Toolkit Intrinsics (Xt). Xt provides a library of user interface objects called widgets and gadgets which can be thought of as the building blocks used to develop a user interface. However, the widgets of Xt are generic and impose no user interface standards whatsoever. What Motif does is to establish the standards for the widgets to provide a common “look and feel” as specified in the *Motif Style Guide* and *Motif Application Environment Specification*. The key point of the specification is that consistency should be maintained across all applications. Similar user interface elements should look and respond similarly regardless of the application.

There are three basic methods that can be employed to achieve this consistency in the user interface:

6. Develop nearly identical features to Motif based on the Xt Toolkit,
7. Develop using the Motif Toolkit,
8. Develop using public domain GUI builders like TCL/TK or commercial user interface products that are based on Motif.

Clearly, number 1 is redundant and unacceptable. Number 2 offers the most flexibility since the full functionality of the toolkit is available, but would require the most effort (outside of 1). Finally, number 3 would provide the fastest mechanism for developing user interfaces since the intent of these products is to streamline the development phase of user interfaces.

7.6 Standard Coding Rules/Practices

This section addresses issues of software style and coding practices. It is intended to provide guidance into the various approaches that bring added value to source code. Among the key issues are readability, comments and methodology. The software developer should constantly consider these issues when writing new code so that others may also benefit.

7.6.1 LIGO Standard Header Comments

The beginning of each LIGO Data Analysis System source file will require a standardized set of descriptive comments which provide a quick look into the origin, description, modification history, etc. A possible template for these Header Comments should be similar to the following in the C language:

```
   /* ----------------------------------------------------------------------------- */
   /*          The Laser Interferometer Gravitation Wave Observatory (LIGO)         */
   /*                          (C) The LIGO Project, 1997                           */
   /*                               ------------------                              */
   /*                        California Institute of Technology                     */
   /*                       Massachusetts Institute of Technology                   */
   /*                               ------------------                              */
   /*   Module Source File:                                                         */
   /*                                                                               */
   /*   Module Description:                                                         */
   /*                                                                               */
   /*   Module Parameters:                                                          */
   /*                                                                               */
   /*   Author Information:                                                         */
   /*         Name:                                                                 */
   /*         Institution:                                                          */
   /*         Support Telephone:                                                    */
   /*         Support E-Mail:                                                       */
   /*                                                                               */
   /*   Coding Specifications:                                                      */
   /*         Standards Conformance:                                                */
   /*         Known Platforms Support:                                              */
   /*         Compiler/Options:                                                     */
```

```
/*                                                                               */
/*   Known Limitations:                                                          */
/*                                                                               */
/* ----------------------------------------------------------------------------- */
```

A similar set of comments should be a part of Fortran and C++ source files.

In addition, the beginning of each source file should have a clean placement of macros and include files. As discussed in the section on POSIX, the user should define the manifested constant _POSIX_SOURCE at the beginning of each source (C or C++) module prior to the inclusion of any headers:

```
/* ----------------------------------------------------------------------------- */
/*                                                                               */
/*   POSIX compliance requires the following:                                    */

#define _POSIX_SOURCE

/*   Header Include File List: (each source file should have a header file)      */

#include <stdio.h>
#include <math.h>
#include <source.h>
/*                                                                               */
/* ----------------------------------------------------------------------------- */
```

Each C or C++ source file should have a corresponding header file. For example the C source file moduleA.c has an associated moduleA.h that contains manifested constants, macros, ANSI function prototypes, structure and union definitions, etc. Similarly for C++ source files with the additional purpose of defining the classes in the associated header file.

7.6.2 Code Comments

The lowest level of Documentation occurs in the source code itself in the form of comment statements. These comments serve to both document the software and to provide understanding to the flow of the program. As previously discussed, there should be a LIGO standard header section in the source code built out of comment statements that present in summary, the software. This should include, but not be limited to, the name of the software, the author, the date created, the purpose and any modifications that have been made, including the name and date of the modification. Header comments add to the understanding of the software when the source code is revisited at a later date or by others for the purpose of fixing bugs, adding new physics, or making general enhancements such as performance and user friendliness.

Comments should also be placed within the body of the code. These comments are to explain the flow of the code, describing the purpose of subroutine and function calls, code blocks and algorithms, and to describe potentially confusing logic. The underlying physics or mathematics should also be referenced in the body, providing a complete conceptual view to the goal of the software

Another important method for commenting code is in the naming of the variables, functions, MACROs and other data types. These names must be descriptive enough to enhance the readability of the code. **It is recommended that these names be full words or concatenations of full words whenever possible**.

7.6.3 Header Macros

The naming of macros in header files should be standardized. For example, the traditional C convention for macros is to use all UPPER_CASE letters and underscores between words. This convention should be adopted. In addition, the use of underscores as the first character in the macro should be strictly prohibited as many C and C++ preprocessors reserve internal macros that begin with either one or two underscore characters.

When developing C++ code, it is standard practice to separate the implementation of the classes from the definition of the classes. The definitions are located in the class header files and the implementations of the class member functions are located in the class source files. This is important for isolating the internal details of the class and allows for better code management. Sometimes it is desirable to include the implementation of a short member function within the definition (*which requests the compiler to attempt to make an inline function call*) to improve performance. **Including the implementation in the definition should be avoided for all but the simplest of functions** since there is no guarantee that the compiler will actually make the function inline.

7.6.4 Magic Numbers

Magic numbers appear in code in two ways. The first is the hard coding of things like number of items, loop counters, index counters, etc. directly into the lines of sources code, the other is the through the use of macros such as #define to set values, sizes, loop counters, etc. The first is by far the most unappealing since all the source code must be searched in order to make a change to the functionality of the software. The second is also unappealing since a change to a value stored in a macro located in a header or include file most be propagated into all the code through a re-compilation of all code that includes the headers. And if the Makefiles are poorly constructed this may not be sensed by the re-compilation without the assistance of the builder. Even in the event that the Makefile does recognize the header as changing, the compilation time can and will be long for large software packages. Magic numbers are basically data in the OOP approach to software and hence should be treated like other data and be initialized at runtime and communicated by messages as much as possible. This approach requires some planning and additional startup overhead in code, but makes code more flexible and easily managed in the long run.

A related topic to magic numbers is the use of enumerated types in C and C++. The use of enumerated datatypes is to be encourages as it adds extra type checking to code and improves the readability.

7.6.5 Header Tags

The inclusion of headers in code needs to be managed by header tags that prevent the header from being multiply included or include loops from being formed. This is most simply managed by having each header file defined a macro variable that can be checked each time the header is included to prevent the preprocessing of the bulk of the header when not necessary. This can greatly speed up the compilation time for poorly written header dependences.

```
/* --------------------------------------------------------------------------- */
/*                                                                             */
/*   Example of Header Tags for header file named myheader.h:                  */

#ifndef MYHEADER_TAG
```

```
/*   Do the preprocessing stuff now and define MYHEADER_TAG                      */

#include <stdio.h>
#include <math.h>
#define MYHEADER_TAG

#endif
/*                                                                               */
/* ----------------------------------------------------------------------------- */
```

This is just a simple example demonstrating one method of achieving the header management. More complicated methods using an if - else - endif structure are not precluded.

7.6.6 Memory Management

The ANSI C standard provides the following functions for dynamic memory allocation:

- void * malloc (const size_t size);
- void * calloc (const size_t num_record, const size_t size_per_record);
- void * free (void * memory_pointer);
- void * realloc (void * old_memory_pointer, const size_t new_size);

These functions provide for efficient memory use in C. In C++ they are replaced by the new and delete operators. The ANSI C functions are still available in C++ but their use in C++ is highly discouraged in favor of the more object oriented connection provided by new and delete.

Virtual memory systems such as Unix reduce the concerns of memory management, but they do not eliminate the need for good practices. It is important to write code that makes efficient use of physical and virtual memory available to the process and the operating system. Software will be written in such a way as to be

- conservative in its use of memory,
- free memory when it is no longer in use,
- use memory in a manner that minimizes the number of memory accesses.

Systems have limited physical memory available. These practices will guarantee that this memory is used in such a way as to optimize efficiency. Excessive use of dynamic memory allocation can lead to poor performance and care must be exercised to insure that memory management is not the source of excess overhead in code while at the same time efficiently making use of a limited resource to the processor.

8.0 Configuration Management

Software Configuration management includes the activities of configuration identification, change control, status accounting, and audits. Baseline software configuration will be provided by the software administrator to LIGO Data Analysis Configuration Control. Pre-baseline development configurations as well as baseline configuration will be controlled by the same software administrator.

8.1 Flow of Configuration Control

The software and documentation developed for the LIGO Data Analysis system will move through four distinct areas, as shown in Figure 2: LIGO DAS Configuration Control Flowchart, to help maintain configuration control. The general flow of software and documentation is:

1. **Development Area:** Area in which software developers work on code in progress. This area has symbolic links and environment variables pointing to the Release Area, to ensure the developer is using the latest released versions of software operating systems and tools.

2. **Archive Area:** Once a developer is satisfied that a particular code is ready for release, the code and documentation is moved into the Archive Area using Concurrent Version System (CVS) commands. Here the code is archived along with necessary history and log records. Code may move back and forth between these first two areas as bugs/faults are detected and repaired, each time being assigned a new version number through CVS. Faults/desired corrections are documented with a Problem / Failure Report (PFR) (discussed later), which travels with the code and is maintained in the Archival Area in the history records.

3. **Release Area:** This is the repository for all source code and documents which have passed test and is ready for build (installation). The assigned release engineer is then responsible for integration of all such code and coordinates the update into the Release Area and the building of the software libraries, object files and executables in the Build Area.

4. **Build Area:** This is where all installed LIGO modeling software are located and made publicly available to users. Once the build area is populated with a new version, a suite of automated tests will be performed to verify reliability.

Further expansion and definition of these areas is covered in the Technical Process section of this document.

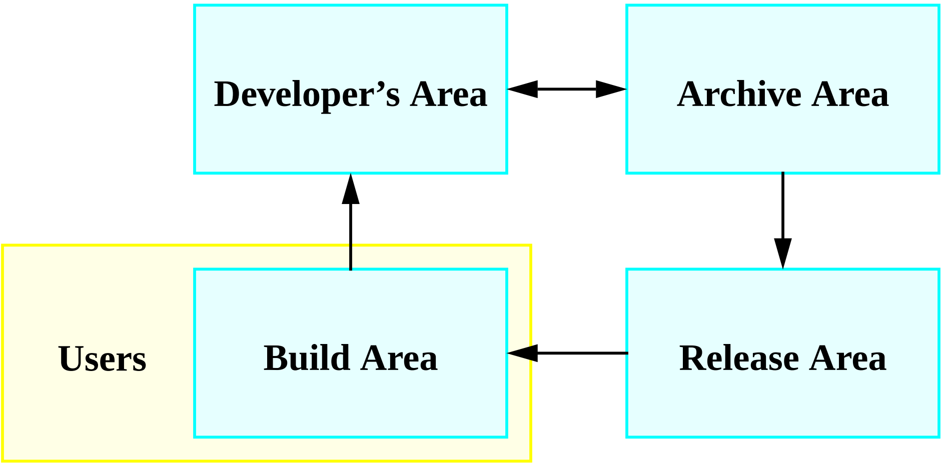

Figure 3: LIGO DAS Configuration Control Flowchart

8.2 Configuration Control Tools

All software in each of the areas described above will always be under source code control using Concurrent Version System (CVS), a powerful front-end to Revision Control System (RCS) or a functionally equivalent product. Both CVS and RCS are distributed under the public domain licensing agreement and available for all Unix platforms.

8.3 Configuration Identification

The configuration identification for each code module will be the revision number assigned automatically by the RCS. Once the Release Engineer has integrated the various code modules, and it has been approved by the software Administrator for release, the integrated code will be put under a unified RCS revision number and released.

The release numbering scheme shall be a three number convention in the form x_x_x, such as 1_2_3. The first number shall indicate a major release. Major releases are typically limited to when the code has undergone major core structural changes or a significant number of enhancements have been made. The second number is changed when a release has new features/enhancements. The final number indicates that bug fixes have been made without the addition of particular features. Developers may choose to implement a four number convention in their local development areas.

9.0 Documentation

The LIGO Data Analysis system must be fully documented. This includes all components from source code to the top level users documentation accessible with Web Browsers.

Graphical User Interfaces (GUI) need to provide through a mechanism similar to menus, the ability to get on-line helps on the functions provided by the GUI.

Unix stand-alone programs for the system should have associated Unix style manpages which are accessible from the shell. In addition, these stand-alone programs should have a command line option such as ‘-help’ or ‘-h’ which presents a quick reference on the proper usage of the program and all of its command line arguments.

All types of libraries that are developed for the Data Analysis system must have a descriptive document listing all the functions provided by the library and the correct usage of each routine with clear examples.

Source code will have an appropriate density of comments that follow the language standard and guide the reader through the flow of the algorithm and logic being performed. This source code shall also have attached to each functional routine the LIGO standard preamble header comments.

A User’s Guide will be provided which gives a complete description of the Data Analysis system and the typical operations of the system. A Developer’s Guide will provide the needed discussions of how to add new components to the system.

The standard language/environment for the documentation needs to be determined. Currently, framemaker, TeX & LaTeX, postscript, pdf, Microsoft Word, HTML and others are extremely popular for generating documentation. Because of the popularity of the World Wide Web and Web browsers such as Netscape, it is absolutely necessary to select documentation environments supporting these Web browsers. At present most do so through translators. Web browsers are also becoming smart, able to recognize document types and spawn readers such as Acrobat Reader for pdf files directly from Netscape.

10.0 Enhancements and Changes (Corrective Action)

Once software has left the development area, all requests for enhancements or corrections are documented in a Problem/Request Report (PRR). A PRR has three basic parts:

1. Problem reporting/enhancement request area submitted by the software's end user.

2. Investigative section, wherein the assigned software engineer investigates the problem/request and provides recommendations to resolve the request.

3. Resolution Area: Information on how the request/problem was resolved.

10.1 Corrective Action Process

Once a PRR is originated, it is submitted to the software Administrator. The Administrator reviews the PRR for appropriateness. If it is not accepted, the initiator is notified. It is then assigned both a priority and a Developer to be responsible for analyzing/resolving the request. Priorities are assigned according to the following table.

Electronic submission methods, initially as a paper copy and electronically entered into a database, will be utilized to guarantee automation, tracking and reliability. This electronic method would be capable of generating reports through the database.

Once the PRR has been analyzed and response returned to the software Administrator, it is reviewed and assigned for implementation. Here it undergoes the same procedures as apply for new software development. Upon completion of test, the PRR is completed by the Developer and returned to the software Administrator for closeout.

Table 2: PRR Priority Assignment

Priority	*Description*
1	The problem adversely affects an essential capability specified in the requirements and no work-around is known.
2	Same as 1 above, but a work-around is known which may be put in place as a temporary solution.
3	The problem causes inconvenience or annoyance but does not affect a requirement.
4	All others not falling into a category above.

From date of origin/receipt until closeout, the status of PRRs will be updated on a weekly basis, with a status page made publicly available such that end users and management can be kept apprised of PRR progress. This status page would be made available to all users through a public mechanism such as the web.

11.0 Reliability

LIGO will collect data with a goal of 90% uptime for individual interferometers over the course of the year. The Data Analysis system will be required to have the same level of reliability or be evaluated to have additional compute performance allowing all analysis to be performed in less than data collection time. The system should be designed with some “over-performance” abilities and at the same time be designed to keep up with the rate of incoming data.

The primary intent for security of the data analysis system is to guarantee reliability and to maximize the availability of the system for carrying out data analysis of LIGO data. It is the goal of LIGO to collect data 24 hours a day for 365 days a year. This goal is aimed at reducing the chances of missing the typical raw short duration gravitational wave event. In order to minimize the expense of the computational hardware base for the LIGO Data Analysis System, the components must have a high level of reliability. Another way of stating this is that the computer system implemented should not have a significant percentage of idle CPU cycles. To achieve this, the system must be robust and designed to tolerate maintenance and down-time. It should also be designed to limit vulnerability from unexpected tampering by “crackers” over the internet. These goals requires systems designed with “hot-swap” components such as hard-drives, CPU boards and power supplies, and “firewalls” to isolate the computational subnets from the general internet. Encryption algorithms should be used to communicate important information and data over wide area networks, such as will exist between the LIGO sites.