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

Technical Note **LIGO-T960004-A - C** 12/1/97

CDS Software Development Plan and Guidelines

R. Bork

Distribution of this draft:

California Institute of Technology
LIGO Project - MS 102-33
Pasadena CA 91125
Phone (818) 395-2966
Fax (818) 304-9834
E-mail: info@ligo.caltech.edu
WWW: http://www.ligo.caltech.edu

1 Introduction 4

1.1. Project Overview 4

1.2. Project Deliverables 4

1.3. Evolution of the SDP 4

1.4. Reference Materials 4

1.5. Definitions and Acronyms 4

1.6. Document Precedence 5

2 Project Organization 5

2.1. Process Model 5

2.2. Organizational Structure 5

3 Management Process 5

3.1. Management Objectives 5

3.2. Risk Management 6

3.3. Personnel Safety and Equipment Protection 6

3.4. Monitoring and Controlling Mechanisms 6
3.4.1. Formal Reviews 6
3.4.2. Informal Reviews 6
3.4.2.1 Design Walkthroughs 6
3.4.2.2 Code Walkthroughs 7
3.4.2.3 Component Informal Reviews 7
3.4.3. Software Configuration Management 7
3.4.3.1 Flow of Configuration Control 7
3.4.3.2 Configuration Control Tools 8
3.4.3.3 Configuration Identification 8
3.4.3.4 Handling of Project Media 8
3.4.3.5 Enhancements and Changes (Corrective Action) 9
3.4.3.5.1 Software Maintenance Request 9
3.4.3.5.2 Corrective Action Process 9
3.4.3.6 Configuration Management Documentation and Reporting .10

4 Technical process 10

4.1. Software Development Life Cycle 10
4.1.1. Conceptual Phase 11
4.1.2. Preliminary Design Phase 11
4.1.3. Final Design Phase 12
4.1.4. First Release Software Development 12
4.1.5. Integration and Commissioning Phase 12

4.2. Design and Development Tools 12
4.2.1. CDS Processors 12

4.2.1.1 Development Hardware 12
4.2.1.2 Target Hardware 12
4.2.2. CASE and Development Tools 12
4.2.3. Programming Languages 13
4.2.4. Documentation Tools 13
4.2.5. Operating Systems 13
4.2.6. Compilers 13
4.2.7. Human-Machine Interfaces 13
4.2.7.1 General Controls 13
4.2.7.2 Additional HMI 13
4.2.8. Data Archival and Retrieval 13
4.2.9. Alarm Management 14
4.2.10. Save and Restore 14
4.2.11. System Diagnostics 14
4.2.12. Application Programmer's Interface 14

4.3. Security 15
4.3.1. Writing files on to CDS systems 15
4.3.2. Interaction with CDS processes 15

4.4. Software Development Guidelines 15
4.4.1. Introduction 15
4.4.2. File Organization 15
4.4.2.1 File Contents 15
4.4.2.2 Source File Layout 16
4.4.2.3 Header File Layout 16
4.4.2.4 Header File Guard 17
4.4.2.5 Prolog 18
4.4.3. Naming Conventions 20
4.4.3.1 EPICS Records 20
4.4.3.2 Conventions within C/C++ Software 21
4.4.3.2.1 Descriptive Names 21
4.4.3.2.2 Valid Characters 21
4.4.3.2.3 File Names 21
4.4.3.2.4 Function Names 21
4.4.3.2.5 Namespaces 21
4.4.4. Style Guidelines 21
4.4.4.1 Lines 22
4.4.4.2 Line Length 22
4.4.4.2.1 Statements Per Line 22
4.4.4.3 Comments 22
4.4.4.3.1 Automated Documentation Comments 22
4.4.4.3.2 Code Block Comments 22
4.4.4.3.3 Blank Lines 22
4.4.4.4 Formatting 22
4.4.4.4.1 Spacing Around Operators 22
4.4.4.4.2 Indentation and Braces 23

4.4.4.4.3 Pointer and Reference Position 24
4.4.4.5 Statements 24
4.4.4.5.1 Control Statements 24
4.4.4.5.2 Conditional Statements 24
4.4.4.5.3 Include Statements 24
4.4.4.6 Declarations 25
4.4.4.6.1 Variable Declaration 25
4.4.4.6.2 External Variable Declaration 25
4.4.4.6.3 Numeric Constant Declaration 26
4.4.4.6.4 Enumerated Type Declaration 26
4.4.4.6.5 Struct and Union Declaration 26
4.4.4.6.6 Class Declaration 26
4.4.5. Recommended Programming Practices 27
4.4.5.1 Placement of Declarations 27
4.4.5.2 Switch Statements 27
4.4.5.3 Return Statements 27
4.4.5.4 Casts 27
4.4.5.5 Literals 27
4.4.5.6 Explicit Initialization 27
4.4.5.7 Constructs to Avoid 28
4.4.5.8 Macros 28
4.4.5.9 Debug Compile-time Switch 28
4.4.5.10 Memory Management 28
4.4.5.11 Constructors 28
4.4.5.12 Destructors 28
4.4.5.13 Argument Passing 29
4.4.5.14 Default Arguments 29
4.4.5.15 Overriding Virtual Functions 29
4.4.5.16 Const Member Functions 29
4.4.5.17 Referencing Non-C++ Functions 29
4.4.5.18 NULL Pointer 29
4.4.5.19 Enumerated Types 29
4.4.5.20 Terminating Stream Output 30
4.4.5.21 Object Instantiation 30
4.4.5.22 Encapsulation 30

1 INTRODUCTION

This Software Development Plan (SDP) describes the software management and development process for the Control and Data System (CDS) for LIGO.

1.1. Project Overview

1.2. Project Deliverables

The project objective is to provide all software as necessary to control and monitor the LIGO systems. This includes the real-time software as necessary to provide automated closed loop control, networking communications to move data in a distributed computing environment, and operator services, such as operator displays, alarm management, slow (10Hz or slower) data archival/retrieval, and system state save and restore capabilities. This does not include high speed (>10Hz) LIGO data acquistion, which will be covered in a separate document.

1.3. Evolution of the SDP

This is intended to be a living, working document over the lifecycle of the project. It will be reviewed for accuracy prior to any formal reviews, whenever higher level LIGO management policies are published to ensure adherence to LIGO standards, and when plan changes are approved which affect this document.

1.4. Reference Materials

1. CDS Control and Monitoring Requirements Document LIGO-T950054-C
2. CDS Control and Monitoring Conceptual Design LIGO-T950120-C

1.5. Definitions and Acronyms

CDS - Control and Data System

CI - Configuration Index

CIM - Computer Integrated Manufacturing

DRR - Design Requirements Review

EPICS - Experimental Physics and Industrial Control System

FDR - Final Design Review

IFO - Interferometer

PDR - Preliminary Design Review

PSL - Pre-Stabilized Laser

SCCS - Source Code Control System

SDL - Software Development Librarian

SMR - Software Maintenance Request

SNL - State Notation Language

SRS - Software Requirements Specification

STP - Software Test Plan

TBD - To Be Determined

1.6. Document Precedence

In the event of conflict between this document and other LIGO documentation, the order of precedence, for this particular project, shall be:

1. LIGO Project Management Plan
2. LIGO Detector Implementation Plan
3. LIGO Project Cost and Schedule Documentation
4. Reference 1
5. This document

2 PROJECT ORGANIZATION

2.1. Process Model

The basic development process model is described in Section 4 of this document. Exact procedures and management processes will be in accordance with the LIGO Detector Implementation Plan. Project schedule and milestones are officially maintained by LIGO Project Management.

2.2. Organizational Structure

LIGO interferometers are to be developed as a team effort within the LIGO Detector Group. This team is made up of members from both the CDS and Interferometer (IFO) sections within the Detector Group. While CDS will be the primary provider of the software, since the final product is the detector itself, all members involved share responsibility in its successful development.

3 MANAGEMENT PROCESS

3.1. Management Objectives

The primary goal of this project is to provide quality software which is an integral part of a Detector Group team effort to provide fully functioning interferometers which meet the requirements of the LIGO detector. Management objectives toward meeting this goal are:

- Early guidance and planning of the project
- Risk Assessment and Analysis
- Incorporating configuration control procedures

- Establishing standard software procedures and coding areas.

3.2. Risk Management

Risk will be analyzed throughout the project lifecycle in terms of technical, cost and schedule risks. Risk analysis shall be presented at each review, along with mitigation techniques.

3.3. Personnel Safety and Equipment Protection

The CDS system will be analyzed from the point of view of personnel safety and machine protection throughout the system lifecycle. Items directly linked to personnel safety will never be resolved in software as the first line of protection. In those cases, hardware will always be the primary safeguard, with software systems only employed in a backup and monitoring role.

Equipment protection may be done in software, depending on the outcome of risk analysis and management decisions. In those cases, the software involved will undergo higher levels of scrutiny during the development and test cycles.

3.4. Monitoring and Controlling Mechanisms

Monitoring and control mechanisms shall be in accordance with LIGO project management plans.

3.4.1. Formal Reviews

All CDS designs proceed through a three stage review process:

- Design Requirements Review (DRR)
- Preliminary Design Review (PDR)
- Final Design Review (FDR)

All reviews are of complete CDS subsystems, which includes electronics as well as the software necessary to meet requirements.

3.4.2. Informal Reviews

The CDS group software developers must work as an integral part of a team with other members of the CDS and the scientists assigned to the LIGO detectors to provide a tightly integrated, functional product. As part of this interaction, informal reviews will be conducted within the LIGO team of scientists and engineers.

3.4.2.1 Design Walkthroughs

This is an informal method used to determine the completeness of a design. The designer conducts the review with attendees representing all affected interfaces. This would also be a forum for the verification of requirements and trade studies.

3.4.2.2 Code Walkthroughs

Verification of design is the primary goal of code walkthroughs. A secondary goal is to check compliance with the CDS adopted software style (TBD), which is important for long term maintenance of the code. Code walkthroughs are particularly important for complicated logic, often found in distributed and/or real-time systems. These walkthroughs will be typically held within the CDS team.

3.4.2.3 Component Informal Reviews

These reviews are frequent, and are intended to allow software to be seen "with a second set of eyes". They occur during work in progress to help verify, at each step, functionality and capabilities of the code, such that a long development has not used up a fair fraction of the schedule prior to a review. They also help ensure that more than one person is familiar with software components.

3.4.3. Software Configuration Management

Software configuration management includes the activities of configuration identification, change control, status accounting, and audits. Baseline software configurations will be provided by the CDS librarian to LIGO Configuration Control. Pre-baseline development configurations as well as baseline configuration will be controlled by the CDS librarian. The approved baseline documents for each component including SRS, STP, and SDP will also be controlled in accordance with LIGO configuration management procedures.

3.4.3.1 Flow of Configuration Control

The software and documentation developed for the LIGO detector systems will move through distinct areas, as shown in the following figure, to help maintain configuration control. The general flow of software and documentation is:

1. Development Area: Area in which software engineers work on code in progress. This area has symbolic links to the Release Area, to ensure the developer is using the latest released versions of software operating systems and tools.
2. Proto/Test Area: Once a developer is satisfied that particular code is ready for release, the code and documentation is moved into a Prototype/Test area. Here the code is integrated and independently tested/operated as part of an overall system. Code may move back and forth between these first two areas as bugs/faults are detected and repaired. Faults/desired corrections are documented with a Software Maintenance Request (SMR) (discussed later), which travels with the code and is maintained in a database to track the history of software.
3. Software Development Librarian (SDL) Area: This is the repository for all code which has passed test and is ready for installation. The assigned librarian is then responsible for integration of all such code and coordinates the update into the Release Area.
4. The Release Area is where all installed LIGO operational systems derive their software.

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

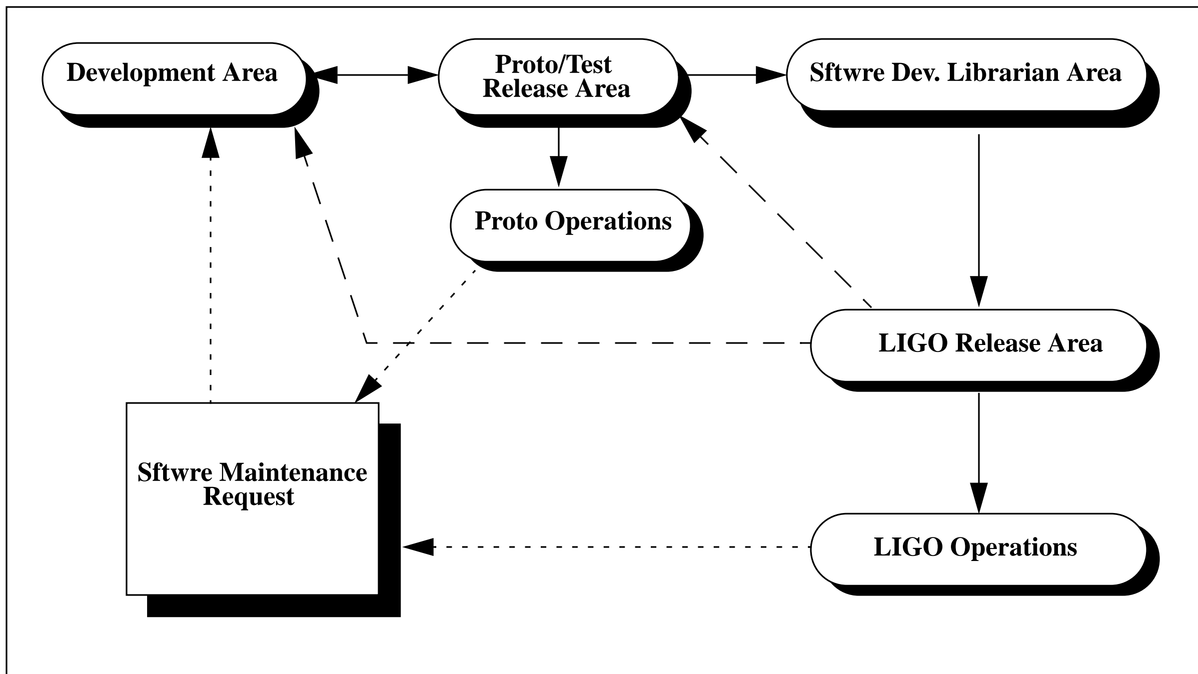


Figure 1: Software Configuration Control Flow

3.4.3.2 Configuration Control Tools

CVS is to be used to provide configuration control.

3.4.3.3 Configuration Identification

The configuration identification for each code module will be the revision number assigned automatically by the CVS. Once the SDL has integrated the various code modules, and it has been approved by the CDS Task Leader for release, the integrated code will be put under a unified CVS revision number and released.

The release numbering scheme shall be a three number convention in the form x.xx, such as 1.23. 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.

3.4.3.4 Handling of Project Media

All CDS documents, source files and build instructions will be locally controlled by the SDL. Whenever a CDS product is approved and baselined (sent to the Release Area), a copy of all materials shall also be turned over to the LIGO Integration Group for LIGO Document Control.

3.4.3.5 Enhancements and Changes (Corrective Action)

3.4.3.5.1 Software Maintenance Request

Once software has left the development area, all requests for enhancements or corrections are documented in an SMR. An SMR has three basic parts:

1. Problem reporting/enhancement request area submitted by the software end user.
2. Analysis section, wherein the assigned software engineer analyzes the problem/request and provides recommendations to resolve the request.
3. Resolution Area: Information on how the request/problem was resolved.

A Mircosoft Access database will be kept of all SMR to help trend and monitor software development projects, which may point to certain areas which may need closer investigation for future software releases. Access forms will be electronically available for submission of SMR, along with Access reports for hardcopies of SMR and database queries.

3.4.3.5.2 Corrective Action Process

Once an SMR is originated, it is submitted to the CDS Task Leader. He/she then assigns both a priority to the SMR and a person to be responsible for analyzing/resolving the request. Priorities are assigned according to the following table.

Table 1: SMR Priority Assignment

Priority	*Description*
1	The problem prevents LIGO from operating to its specified performance as a detector. The problem jeopardizes personnel safety or machine protection.
2	The problem adversely affects either an essential capability specified in the requirements or the operator's accomplishment of that capability, and no work-around is known.
3	Same as 2 above, but a work-around is known which may be put in place as a temporary solution.
4	The problem causes inconvenience or annoyance but does not affect a requirement.
5	All others not falling into a category above.

In the event an SMR is a request for enhancements or change in project scope, the CDS Task Leader will determine if this request must be processed through the LIGO Change Control System prior to further assignment to a software engineer.

Once the SMR has been analyzed and response returned to the CDS task leader, it is reviewed and assigned for implementation. Here it undergoes the same procedures as apply for new software

development. Upon completion of test, the SMR is completed by the developer and returned to the CDS Task Leader for closeout.

From date of origin/receipt until closeout, the status of SMR's 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 SMR progress.

3.4.3.6 Configuration Management Documentation and Reporting

The primary reporting will be in the form of a Configuration Index (CI). During the design phase, software components will be identified, which are then tracked thoughout the software lifecycle. One CI is prepared for each of these components. The CI includes an historical record section, milestone data, list of associated documentation, and a list of applicable SMR or other project change requests (including status/disposition).

4 TECHNICAL PROCESS

4.1. Software Development Life Cycle

Software development will follow the standard waterfall life cycle as much as possible. This cycle is shown in Figure 2: Software Development Cycle. While this outlines the general flow of development, reiteration between certain phases will occur, for instance, prototype and test may indicate that requirements need to be changed/updated or new approach taken.

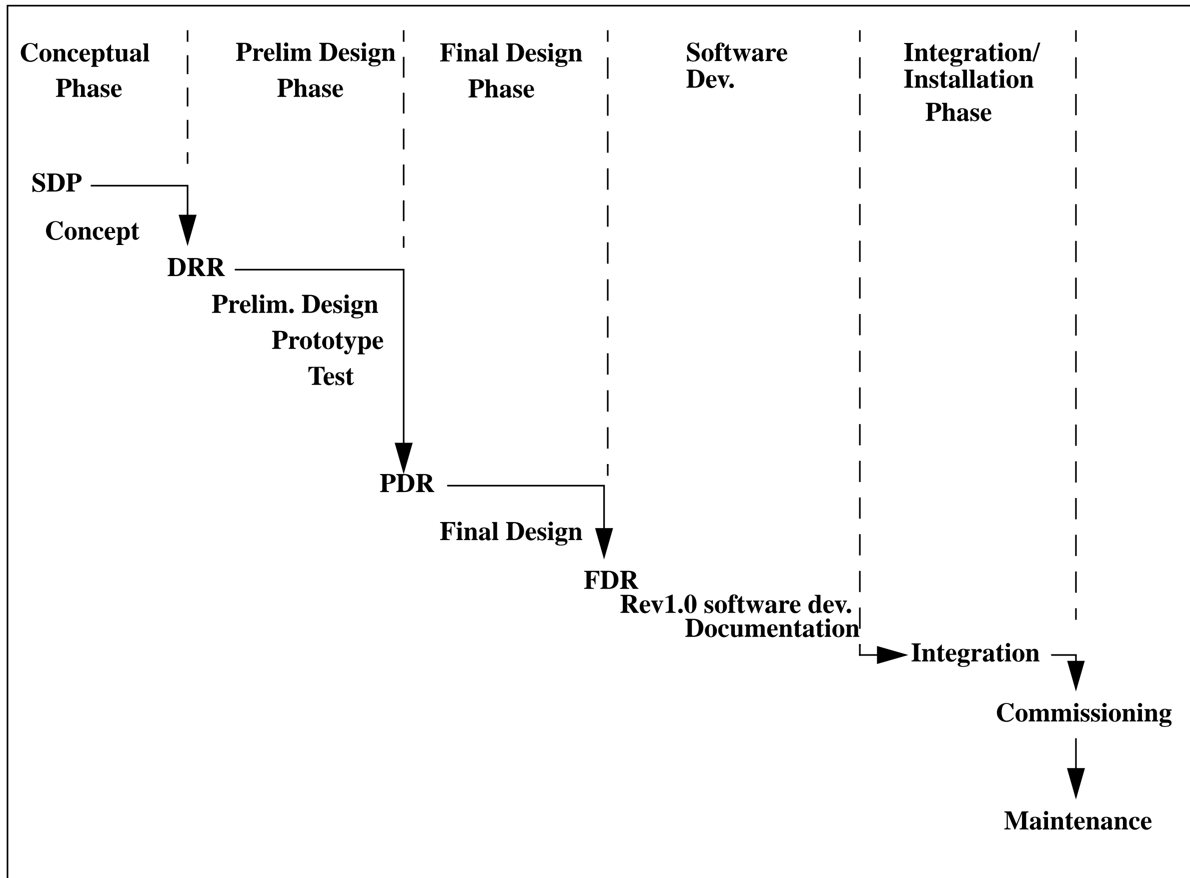


Figure 2: Software Development Cycle

4.1.1. Conceptual Phase

During the conceptual phase, a Design Requirements Document (DRD) is developed for each LIGO CDS subsystem. This DRD is for the entire subsystem and does not differentiate between electronics/hardware and software. A subsystem conceptual design is also developed from these requirements and documented in a conceptual design document. This phase ends with a formal Design Requirements Review (DRR).

4.1.2. Preliminary Design Phase

During the premininary design phase, software and electronic engineers will work closely to develop an overall system design, determining which parts of the system will be implemented in hardware and which in software. Components of the resulting design which are determined to be technical risks are then prototyped during this design phase. The resulting designs and prototype tests are then documented in a preliminary design document, which is formally reviewed at the Preliminary Design Review.

4.1.3. Final Design Phase

In this phase, any outstanding design issues from the PDR are resolved, and final code designs are documented. The subsystem under design is then reviewed, along with any electronics, in a Final Design Review.

4.1.4. First Release Software Development

During this phase, a Release 1.0 software package is developed. Release 1.0 is defined as that which meets the minimum requirements to begin installation and commissioning of LIGO. Release 1.0 shall contain:

1. Software Test Plans (STP) which exercise the software and ensure all requirements are met.
2. Code necessary to allow commissioning of the system.
3. Supporting documentation of successful completion of testing in accordance with the STP.
4. The first set of software system documentation.

4.1.5. Integration and Commissioning Phase

Software integration and commissioning will proceed through various stages:

1. When moved from the development to the test areas, initial integration will take place.
2. Second level integration and commissioning will occur, where possible, on any systems deployed to LIGO prototypes, such as the 40 meter lab.
3. If schedule permits, full up CDS systems for the interferometers will be assembled within the electronics shop areas at the LIGO sites. Here, as much integration and pre-commissioning as possible will take place prior to final installation.
4. Upon final installation, commissioning will take place in accordance with a LIGO commissioning plan.

4.2. Design and Development Tools

4.2.1. CDS Processors

4.2.1.1 Development Hardware

The software development hardware will be Sun workstations.

4.2.1.2 Target Hardware

The project involves two types of target hardware:

1. Sun workstations for operator and file services.
2. VME-based processors for real-time control applications.

4.2.2. CASE and Development Tools

The CDS software will be designed and developed based on the present capabilities of the Experimental Physics and Industrial Control System (EPICS) software package, distributed by Los Alamos National Laboratory (LANL). For this project, EPICS will be used “as is” for the bulk of the

activities, except to address those issues where use of EPICS is analyzed as not being able to meet particular CDS requirements.

4.2.3. Programming Languages

In those cases were EPICS will not meet a requirement, or other new software needs to be developed, the ‘C’ language will be used, designed with standard software design tools.

4.2.4. Documentation Tools

All documentation will be produced with the LIGO standard publishing packages and tracked through the CDS database. When approved, this documentation will be reproduced on the World Wide Web.

4.2.5. Operating Systems

Two computer operating systems will be employed in this project:

1. Sun Solaris for code development on Sun workstation targets.
2. VxWorks real-time operating system for all VME based processor targets.

4.2.6. Compilers

Compilers for C and C++ will be provided. However, for direct connection of software to CDS real-time data, only C code will be supported. This is due to the fact that all data interface routines presently available to provide data connections to EPICS channel access are written in C. There are no plans to provide C++ or Fortran versions of this API software.

4.2.7. Human-Machine Interfaces

4.2.7.1 General Controls

For purposes of both developing HMI and providing the primary interactive runtime HMI, the SAMMI product from Kinesix and the standard EPICS extension Motif Editor and Display Manager (MEDM) will be supported.

4.2.7.2 Additional HMI

Additional HMI will be supported for various purposes. At present, xmgr is used for plot displays, as an example. As analysis software and other special purpose software is developed within LIGO, the CDS will incorporate and support those packages which become defacto standards.

4.2.8. Data Archival and Retrieval

The bulk of the LIGO data will be acquired and archived by the DAQS, as described in LIGO T970136-00-C. The control and monitoring systems will be provided access to this data and data archival/retrieval software tools via CDS networks.

4.2.9. Alarm Management

Alarm enunciation, display and logging will be provided using the EPICS alarm manager (ALM). ALM allows for:

- The definition and structuring of alarm trees via an ascii editor using ALM keywords and guidelines.
- Alarm enunciation and display of the alarm tree.
- Alarm logging and playback.
- Defining and displaying operator guidance along with the alarm states.
- Defining and allowing operator execution of real-time processes to deal with alarm conditions.

4.2.10. Save and Restore

Save and restore provides the capability to take “snapshots” of CDS control settings/readings to allow resetting control parameters to the same configuration at a later time. The Back-Up and Restore Tool (BURT) of EPICS will be used to provide this functionality.

BURT provides:

- Collection and storage to user defined files of system setpoints and readings. Data to collect is defined by the user in ascii files using BURT keywords and structures.
- Resetting of setpoint parameters on demand from the operator.
- Viewing and modification capabilities prior to resetting values.
- Concatenation of multiple back-up files.
- Basic math routines to adjust back-up settings prior to resetting the real-time systems.

4.2.11. System Diagnostics

The initial set of diagnostics will be those provided with the VxWorks and Unix operating systems, along with the EPICS tools. The EPICS tools include:

- VxWorks command line interrogation, such as listing of records, records attached by SNL code, and status of I/O drivers.
- Probe: An X window tool which provides display of values from EPICS record fields.

In the longer term, GUI interfaces will be built onto the system to provide:

- Status of all CDS software modules.
- Status of all CDS I/O modules.
- Status of all CDS networks.
- Status of all CDS mass storage systems.

4.2.12. Application Programmer’s Interface

To provide connection of CDS data to code developed by other user’s, the primary API will be the CA call libraries and the EZCA libraries. Both provide embeddable C calls to allow access to EPICS data via CA. CA libraries provide the most versatility in asynchronous callbacks, but require a higher level of programming skills. EZCA provides easier to use function calls, but is more limited in its capabilities. Instructions for use of these libraries are provided in the EPICS manual set.

4.3. Security

4.3.1. Writing files on to CDS systems

Routine log-in procedures requiring user identification and password provides access control to CDS software areas. These areas will be open to all LIGO personnel for data “read”, but will be write accessable only by the designated CDS group members (code release areas are restricted to the CDS software librarian). At minimum, weekly backups of all files will be maintained in order to prevent catastrophic loss of data.

4.3.2. Interaction with CDS processes

The design of the CDS allows setpoint adjustments and readback information to be accessed from all processors connected to the CDS network. To prevent unauthorized/inadvertent adjustments to the control system from other processing systems, only CDS computers will be connected directly to this network. Information (read only) will be provided to systems outside of the CDS network via a CDS/General computing firewall computer. This machine will require remote login and passwords and will only be authorized to provide monitoring information. This computer will not be privileged to allow adjustments to interferometer parameters or software.

4.4. Software Development Guidelines

4.4.1. Introduction

The purpose of these coding standards is to facilitate the maintenance, portability, and reuse of custom C and C++ source code developed for LIGO CDS systems. Most of the standards in this section were taken directly from the *Coding Standards for C, C++, and Java* developed by the Vision 2000 CCS Package and Application Team. Additions are primarily made to the Naming Conventions and Prolog sections.

4.4.2. File Organization

4.4.2.1 File Contents

Files should be used to organize related code modules, either at the class (for C++) or function(for C) level. The following table identifies the contents of individual files for each language:

Table 2:

File Contents	*C*	*C++*
class declaration (header)	n/a	X
class definition (source)	n/a	X
main function	X	X
function(s)	X	X

Table 2:

File Contents	*C*	*C++*
globals	X	X

4.4.2.2 Source File Layout

Source files should contain the following components in the order shown:

File contents C C++

Table 3:

File Contents	*C*	*C++*
prolog	X	X
package imports	n/a	X
system #includes	X	X
application #includes	X	X
external functions	X	X
external variables	X	X
constants	X	X
static variable initializations	X	X
public methods	n/a	X
protected methods	n/a	X
private methods	n/a	X
functions	X	X

When its possible to put a needed #include line in the source file instead of in the header file, do so. This will reduce unnecessary file dependencies and save a little compile time.

4.4.2.3 Header File Layout

Header files should contain the following components in the order shown (note that Java does not use header files):

Table 4:

File Contents	*C*	*C++*
file guard	X	X

Table 4:

File Contents	*C*	*C++*
prolog	X	X
system #includes	X	X
application #includes	X	X
#defines	X	X
macros	X	X
external functions	X	X
external variables	X	X
constants	X	X
structs	X	X
forwrd declarations	X	X
class declaration	n/a	X
public methods	n/a	X
protected methods	n/a	X
private methods	n/a	X
inline method definitions	n/a	X
functions	X	X

(C++) Small inline methods may be implemented in the class definition.

4.4.2.4 Header File Guard

- *(C,C++)* All header files should contain a file guard mechanism to prevent multiple inclusion. This mechanism is implemented as shown by the following lines:

```
#ifndef MeaningfulNameH              // first line of the header file
#define MeaningfulNameH              // second line of the header file
      .
      .                              // body of the header file
.
#endif // MeaningfulNameH            // last line of the header file; note comment
```

4.4.2.5 Prolog

All software developed for CDS systems shall have a prolog which follows the format shown on the following page. This prolog is to include:

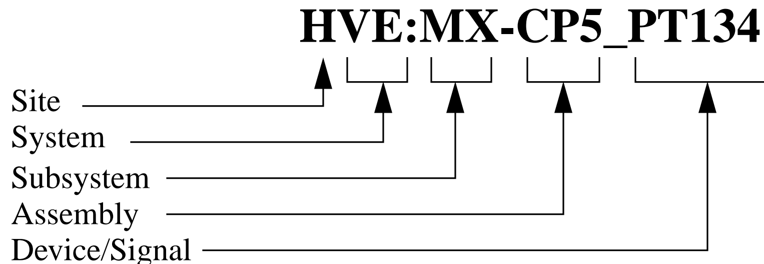
1. Code module name
2. A brief description of the software and its function.
3. Any arguments required.
4. A revision history.
5. List of documentation references.
6. Author information.
7. Code compilation and runtime specifications, including:
   - Compilation information
   - Runtime target information
   - Any additional software (not included within this code file) necessary to link/run the software.
8. Checkoff list that code has met standards: lint, ASCII, POSIX; also, if this software does not meet standards, comments stating the reasons should be included in this section.
9. Known bugs, limitations, cavaets
10. Copyright information

```
/*-----------------------------------------------------------------------------          */
/*                                                                                        */
/* Module Name: framew.c                                                                  */
/*                                                                                        */
/* Module Description: 40m Data Acquisition System.                                       */
/*                     FrameBuilder (FB)        .                                         */
/*                                                                                        */
/* Module Arguments:                                                                      */
/*                                                                                        */
/* Revision History:                                                                      */
/* Rel   Date    Engineer   Comments                                                      */
/* 1.0   14Mar97 R. Bork    First Release.                                                */
/*                                                                                        */
/* Documentation References:                                                              */
/*      Man Pages:                                                                        */
/*      References:                                                                       */
/*                                                                                        */
/* Author Information:                                                                    */
/*      Name         Telephone    Fax          e-mail                                     */
/*      Rolf Bork   . (818)3953182 (818)5440424 rolf@ligo.caltech.edu                     */
/*                                                                                        */
/* Code Compilation and Runtime Specifications:                                           */
/*      Code Compiled on: Sun Ultra Enterprise 2 running Solaris2.5.1                     */
/*      Compiler Used: Heurikon's gcc-sde                                                 */
/*      Runtime environment: Baja47 running VxWorks 5.2 Beta B.                           */
/*      Additional code objects  required: fb.db fb.o FrameL.o                            */
/*                                                                                        */
/* Code Standards Conformance:                                                            */
/*      Code Conforms to: LIGO standards.       OK                                        */
/*                        Lint.                 TBD                                       */
/*                        ANSI                  TBD                                       */
/*                        POSIX                 TBD                                       */
/*                                                                                        */
/* Known Bugs, Limitations, Caveats:                                                      */
/*      1) Only frames 17 fast ADC channels due to network limits                         */
/*      2) No timestamps pending receipt of GPS                                           */
/*      3) Not all frame fields filled in, only data for now                              */
/*                                                                                        */
/*                      -------------------                                               */
/*                                                                                        */
/*                            LIGO                                                        */
/*                                                                                        */
/*  THE LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY.                              */
/*                                                                                        */
/*                  (C) The LIGO Project, 1996.                                           */
/*                                                                                        */
/*                                                                                        */
/* California Institute of Technology                                                     */
/* LIGO Project MS 51-33                                                                  */
/* Pasadena CA 91125                                                                      */
/*                                                                                        */
/* Massachusetts Institute of Technology                                                  */
/* LIGO Project MS 20B-145                                                                */
/* Cambridge MA 01239                                                                     */
/*                                                                                        */
/*-----------------------------------------------------------------------------          */
```

4.4.3. Naming Conventions

4.4.3.1 EPICS Records

All EPICS database records used within LIGO systems are to have a unique name based on the following standard guidelines.



Examples are:

H1:PSL-FSS_PC112 (Hanford IFO 1 : PSL subsystem - Freq. Stab. Servo _ Signal Name

HVE:MX-CP5_PT134 (Hanford Vacuum Equip. : X arm mid station subsystem - Cryopump 5 _ pressure transducer 134

Where:

1. First field (prior to ‘:’) is the site/system designator:
- First character is a letter designating the site:
    - H for Hanford
    - L for Livingston
    - C for Caltech
    - M for MIT
- Next character set represents the system:
    - A number (1-6) for an interferometer
    - VE, for vacuum equipment system
    - PM, for PEM system
- Third character is a ‘:’ (colon), used as a division character for name searches.
2. Second field (preceded by ‘:’ and ended with ‘-’) is the subsystem designator. The following are examples:
    - MX = X arm mid station
    - PSL = PreStabilized Laser
    - LSC = Length Sensing & Control
    - ASC = Alignment Sensing & Control
3. Third field is the assembly designator (preceded by ‘-’ and terminated in ‘_’. In the examples given FSS would be the Frequency Stabilization Servo and CP5 would be cryopump number 5.
4. Final field is the signal designator. In the second example, PT134 is pressure transducer 134.

4.4.3.2 Conventions within C/C++ Software

4.4.3.2.1 Descriptive Names

Names should be readable and self-documenting. Abbreviations and contractions are discouraged. Shorter synonyms are allowed when they follow common usage within the domain.

4.4.3.2.2 Valid Characters

All names should begin with a letter. Individual words in compound names are differentiated by capitalizing the first letter of each word as opposed to separating with an underscore. The use of special characters (anything other than letters and digits), including underscores is strongly discouraged.

4.4.3.2.3 File Names

Filenames should only contain one period, to separate the file extension. Reserved file extensions are:

```
.c        // C source code
.cpp      // C++ source code
.o        // Object files
.h        // Header files
.db       // EPICS database files
.alh      // EPICS alarm handler files
.adl      // EPICS MEDM files
.dctsdr   // EPICS database drivers
.st       // EPICS State Notation Language source code
```

4.4.3.2.4 Function Names

Function names should preferably be an action verb. Boolean-valued functions (those that have two possible return values) should use the "is" prefix as in "isEmpty()".

All functions must be prototyped, with the prototypes residing in header files.

4.4.3.2.5 Namespaces

Namespace collision should be minimized without introducing cryptic naming conventions by using the C++ namespace.

4.4.4. Style Guidelines

The primary purpose of style guidelines is to facilitate long-term maintenance. During maintenance, programmers who are usually not the original authors are responsible for understanding source code from a variety of applications. Having a common presentation format reduces confusion and speeds comprehension. Therefore, the following guidelines are specified based on the principles of good programming practice and readability. In the cases where two or more equally valid alternatives are available, one was selected to simplify specification. In the future, automated tools may be used to apply style guidelines to source code files.

4.4.4.1 Lines

4.4.4.2 Line Length

All lines should be displayable without wrapping on an 80-character display. If wrapping is required, try to break at an operator, and start the next line with the operator vertically aligned. For example:

```
cout << "This is an example of a line which must be wrapped, value = "
     << value << endl;
```

4.4.4.2.1 Statements Per Line

Each statement should begin on a new line.

4.4.4.3 Comments

4.4.4.3.1 Automated Documentation Comments

For comments meant to be extracted by an automated documentation tool, follow the Java convention of using the standard C comment delimiters with an extra asterisk on the first one, as shown:

```
/**
* This is a module, class, function, or instance variable comment
* that will be extracted by an automated documentation tool.
*/
```

This will provide a consistent look across all source code files, and should facilitate creation ofautomated documentation tools. Such comments should be used to describe classes, methods, and global or instance variables.

4.4.4.3.2 Code Block Comments

Code block comments should precede the block, be at the same indentation level, and be separated by a blank line above and below the comment. Brief comments regarding individual statements may appear at the end of the same line, and should be vertically aligned with other comments in the vicinity for readability.

- *(C)* Code block comments should use the standard C comment delimiters /* and */.
- *(C++)* Code block comments should use the single line comment delimiter //.

4.4.4.3.3 Blank Lines

Use a single blank line to separate logical groups of code to improve readability. In source files, use two blank lines to separate each function.

4.4.4.4 Formatting

4.4.4.4.1 Spacing Around Operators

Spacing around operators and delimiters should be consistent. In general, insert one space before or after each operator to improve readability. Use spaces inside of the parentheses around the argument list. Do not use a space within empty argument lists () or non-dimensioned arrays [].

- *(C++)* Do not use spaces around the scope operator ::.
- *(C++)* Do not use spaces around the member access operators . and ->.

```
if ( value == 0 ) {                 // right
if (value==0){                      // not recommended

void doIt( int v );                 // right
void doIt(int v);                   // not recommended

value = object->GetValue();         // right
value=object -> GetValue();         // wrong
```

4.4.4.4.2 Indentation and Braces

The contents of all code blocks should be indented to improve readability. A single tab or four spaces are recommended as the standard indentation. Braces should be placed to show the level of indentation of the code block, with the open brace at the end of the statement which starts the block, and the close brace indented to match the statement.

```
int main() {
    doSomething();
}

struct MyStruct {
    int x;
    int y;
}

if ( value == 0 ) {
    doSomething();
} else if ( value == 2 ) {                  // note position of cascaded if statement
    doSomething2();
} else {
    doSomething3();
}

while ( value < 300 ) {
    doSomething();
}

do {
    doSomething();
} while ( value < 300 )                     // note: ending brace and control on same line

switch ( value ) {
case 1:
    doSomething();
    break;
case 2:
case 3:
    doSomething2();
    break;
default:
    break;                                  // final break required
}
```

4.4.4.4.3 Pointer and Reference Position

All declarations of pointer or reference variables and function arguments should have the dereference operator * and the address-of operator & placed adjacent to the type, not the variable. For example:

```
char* text;                          // right
char *text;                          // not recommended

char* doSomething( int* x );         // right
char *doSomething( int *x );         // not recommended
```

4.4.4.5 Statements

4.4.4.5.1 Control Statements

All control statements should be followed by an indented code block enclosed with braces, even if it only contains one statement. This makes the code consistent and allows the block to be easily expanded in the future. For example:

```
if ( value == 0 ) {                          // right
        doSomething();
}

if ( value == 0 ) doSomething();             // wrong - no block, not indented

if (value == 0)
        doSomething();                       // wrong - no block
```

4.4.4.5.2 Conditional Statements

Conditional statements found in if, while, and do statements should be explicit based on the data type of the variable being tested. For example:

```
int value = getValue();
if ( value == 0 ) {                          // right
        doSomething();
}
if ( !value ) {                              // wrong - not explicit test
        doSomethingElse();
}
bool value = getValue();                     // could be RWBoolean too.

if ( !value ) {                              // right
        doSomethingElse();
}
```

4.4.4.5.3 Include Statements

For both source and header files, #include statements should be grouped together at the top of the file after the prolog. Includes should be logically grouped together, with the groups separated by a blank line. System includes should use the <file.h> notation, and all other includes should use the "file.h" notation. Path names should never be explicitly used in #include statements (with the exception of vendor library files such as Motif), since this is inherently non-portable. For example:

```
#include &ltstdlib.h>                       // right
#include &ltstdio.h>                        //
#include &ltXm/Xm.h>                        //
#include "meaningfulname.h"                 //
#include "/proj/util/MeaningfulName.h"      // wrong - explicit path,
#include &ltstdlib.h>                       // out of order,
#include </usr/include/stdio.h>             // path for system file,
#include "Xm/Xm.h"                          // local include of library file
```

4.4.4.6 Declarations

4.4.4.6.1 Variable Declaration

Each variable should be individually declared on a separate line. Variables may be grouped by type, with groups separated by a blank line. Variable names should be aligned vertically for readability. There is no required ordering of types, however some platforms will give optimal performance if declarations are ordered from largest to smallest (e.g., double, int, short, char).

```
int* a;                                     // right
int b;                                      //
int c;                                      //
double d;                                   //
double e;                                   //
double a;                                   // right
int b;                                      //
double d;                                   // acceptable - not grouped by type
int b;                                      //
int* a;                                     //
double e;                                   //
int c;                                      //
int* a, b, c;                               // wrong - not individually declared, not
                                            // on separate lines
int* a,                                     // wrong - not individually declared
b,                                          //
c;                                          //
```

The two preceding examples are prone to error; notice that a is declared as a pointer to integer and b and c are declared as integers, not as pointers to integers.

4.4.4.6.2 External Variable Declaration

All external variables should be placed in header files. In general the use of global variables is discouraged. Use the following method to allow external variables to be created only once while using a single declaration. In the header file which declares the global variable, use a flag to cause the *default* action on inclusion to be referencing of an externally created variable. Only in the source file that wants to actually create the variable will this flag be defined.

In the header file MeaningfulName.h,

```
#ifdef MeaningfulNameInit                   // the flag is called MeaningfulNameInit
#define EXTERN                              // create the variable (only in main.cpp)
#else
#define EXTERN extern                       // just a reference (default)
#endif
EXTERN ErrorLogger errorLog;
#undef EXTERN
```

All of the source files should include this header file normally:

```
#include meaningfulname.h
```

while the following should appear *only* in the source file where you actually want to declare the variable and allocate memory for it (typically in main.cpp):

```
#define MeaningfulNameInit
#include meaningfulname.h
#undef MeaningfulNameInit
```

4.4.4.6.3 Numeric Constant Declaration

Use only the uppercase suffixes (e.g., L, X, U, E, F) when defining numeric constants. For example:

```
const int value = A73B2X;          // right, hexadecimal constant
const double evalue = 1.2E9;       // right, scientific notation constant
const float fvalue = 1.2e9;        // wrong, lowercase e
```

4.4.4.6.4 Enumerated Type Declaration

(C++) The enum type name and enumerated constants should each reside on a separate line. Constants and comments should be aligned vertically. Following is an example of a valid enum declaration:

```
enum CompassPoints {               // Enums used to specify direction.
North = 0,                         //
South = 1,                         //
East = 2,                          //
West = 3                           //
};
```

4.4.4.6.5 Struct and Union Declaration

The struct type name and structure members should each reside on a separate line. This format separates the members for easy reading, is easy to comment, and eliminates line wrapping for large numbers of members. Each struct should have a one-line description on the same line as the type name. Each member should have a comment describing what it is, and units if applicable. Members and comments should be aligned vertically. Following is an example of a valid struct declaration:

```
struct MeaningfulName {            // This is a struct of some data.
int firstInteger;                  // This is the first int.
int secondInteger;                 // This is the second int.
double firstDouble;                // This is the first double.
double secondDouble;               // This is the second double.
};
```

4.4.4.6.6 Class Declaration

(C++) All class definitions should include a constructor, (virtual) destructor, copy constructor and operator=. If the class has a pointer, provide a deep copy constructor (i.e., allocates memory and copies the object being pointed to, not just maintains a pointer to the original object). If any of these four are not currently needed, create stub versions and place in the private section so they

will not be automatically generated, then accidentally used. (This protects from core dumps.) All classes should have public, protected, and private access sections declared, in this order. Friend declarations should appear before the public section. All member variables should be either protected or private. It is recommended that definitions of inline functions follow the class declaration, although trivial inline functions (e.g., {} or { return x;

```
void incrementValue();          // Increment value.
private:
int value;                      // The value.
};
```

4.4.5. Recommended Programming Practices

4.4.5.1 Placement of Declarations

Local variables can be declared at the start of the function, at the start of a conditional block, or at the point of first use. However, declaring within a conditional block or at the point of first use may yield a performance advantage, since memory allocation, constructors, or class loading will not be performed at all if those statements are not reached.

4.4.5.2 Switch Statements

Specify a break statement after every case block, including the last one unless multiple labels are used for one selection of code. It is recommended that a default case always be defined.

4.4.5.3 Return Statements

Where practical, have only one return from a function or method as the last statement. Otherwise, minimize the number of returns. Possibly highlight returns with comments and/or blank lines to keep them from being lost in other code. Multiple returns are generally not needed except for reducing complexity for error conditions or other exceptional conditions.

4.4.5.4 Casts

Avoid the use of casts except where unavoidable, since this can introduce run-time bugs by defeating compiler type-checking. Working with third-party libraries (e.g., X or Motif) often requires the use of casts. When you need to cast, document the reasons.

4.4.5.5 Literals

Use constants instead of literal values wherever possible. For example:

```
const double PI = 3.141259;                        // right
const char APP_NAME = "ACME Spreadsheet 1.0";      // right
area = 3.141259 * radius * radius;                 // not recommended
cout << "ACME Spreadsheet 1.0" << endl;            // not recommended
```

4.4.5.6 Explicit Initialization

In general, explicitly initialize all variables before use.

It is very strongly recommended that you initialize all pointers either to 0 or to an object. Do not allow a pointer to have garbage in it or an address in it, that will no longer be used.

4.4.5.7 Constructs to Avoid

The use of #define constants is strongly discouraged, using const is recommended instead.

The use of #define macros is strongly discouraged, using inline functions is recommended instead.

The use of typedef is discouraged when actual types such as class, struct, or enum would be a better choice.

The use of extern (e.g., global) variables is strongly discouraged. The exception is for programs which benefit from having a small number of object pointers accessible globally via extern. The use of goto statements is not allowed.

4.4.5.8 Macros

All arguments to macros should be enclosed in parentheses to eliminate ambiguity on expansion. For example:

```
#define MAX( x, y ) ( (x) > (y) ) ? (x) : (y) )
```

4.4.5.9 Debug Compile-time Switch

Code used only during development for debugging or performance monitoring should be conditionally compiled using #ifdef compile-time switches. The symbols to use are DEBUG and STATS, respectively. Debug statements announcing entry into a function or member function should provide the entire function name including the class. For example:

```
#ifdef DEBUG
        cout << "MeaningfulName::doSomething: about to do something" << endl;
#endif
```

4.4.5.10 Memory Management

(C++) Use new and delete instead of malloc/calloc/realloc and free. Allocate memory with new only when necessary for variable to remain after leaving the current scope. Use the delete [] operator to deallocate arrays (the use of delete without the array operator to delete arrays is undefined). After deletion, set the pointer to zero, to safeguard possible future calls to delete. C++ guarantees that delete 0 will be harmless.

4.4.5.11 Constructors

(C++) All constructors should initialize all member variables to a known state. This implies that all classes should have a default constructor (i.e., MyClass();) defined. Providing a deep copy constructor is strongly recommended. If the programmer wishes to not fully implement a copy constructor, then a stub copy constructor should be written and placed in the private section so no one will accidentally call it.

4.4.5.12 Destructors

(C++) All classes which allocate resources which are not automatically freed (e.g., have pointer variables) should have a destructor which explicitly frees the resources. Since any class may someday be used as a base class, destructors should be declared virtual, even if empty.

4.4.5.13 Argument Passing

(C++) If the argument is small and will not be modified, use the default pass by value. If the argument is large and will not be modified, pass by const reference. If the argument will be modified, pass by reference. For example:

```
void A::function( int notChanged );                  // default: pass by value
void B::function( const C& bigReadOnlyObject )       // pass by const reference
void C::function( int notChanged, int& result );     // pass by reference
```

4.4.5.14 Default Arguments

(C++) Where possible, use default arguments instead of function overloading to reduce code duplication and complexity.

4.4.5.15 Overriding Virtual Functions

(C++) When overriding virtual functions in a new subclass, explicitly declare the functions virtual. Although not required by the compiler, this aids maintainability by making clear that the function is virtual without having to refer to the base class header file.

4.4.5.16 Const Member Functions

(C++) It is recommended that all member functions (example: func(...) const {...}) which do not modify the member variables of an object be declared const. This allows these functions to be called for objects which were either declared as const or passed as const arguments.

(C++) It is recommended that all member function parameters be declared const (example: func(const int i){...}) when possible.

4.4.5.17 Referencing Non-C++ Functions

(C++) Use the extern "C" mechanism to allow access to non-C++ (not just C) functions. This mechanism disables C++ name mangling, which allows the linker to resolve the function references. For example:

```
extern "C" {
void aFunction();                   // single non-C++ function prototype
}
extern "C" {
#include "functions.h"              // library of non-C++ functions
}
```

4.4.5.18 NULL Pointer

(C++) Use the number zero (0) instead of the NULL macro for initialization, assignment, and comparison of pointers. The use of NULL is not portable, since different environments may define it to be something other than zero (e.g., (char*)0).

4.4.5.19 Enumerated Types

(C++) Use enumerated types instead of numeric codes. Enumerations improve robustness by allowing the compiler to perform type-checking, and are more readable and maintainable.

4.4.5.20 Terminating Stream Output

(C++) Use the iostream manipulator endl to terminate an output line, instead of the newline character \n. In addition to being more readable, the endl manipulator not only inserts a newline character but also flushes the output buffer.

4.4.5.21 Object Instantiation

(C++) Where possible, move object declarations and instantiations out of loops, using assignment to change the state of the object at each iteration. This minimizes overhead due to memory allocation from the heap.

4.4.5.22 Encapsulation

(C++,Java) Instance variables of a class should not be declared public. Open access to internal variables exposes structure and does not allow methods to assume values are valid.

(C++) Putting variables in the private section is preferable over the protected section, for more complete encapsulation. Use get and set methods in either protected or public if needed.

LIGO CDS Maintenance Request Form

Bug/Change Request No.: __________

Problem Report.

Raised by: ________________________ Date: ________________ System: __________ Sub-system: __________

Problem is: hardware-bug ☐ software-bug ☐ change-request ☐ new-requirement ☐ other ☐

Description:

Priority:
- ☐ 1) Safety issue or prevents Ligo operation
- ☐ 2) Affects essential capability, no work-around available
- ☐ 3) Affects essential capability, temporary work-aound available
- ☐ 4) Inconvenience/annoyance
- ☐ 5) Other (Minor)

Analysis.

Done by: ________________________ Date: ________________

Impact on other systems:____

Estimated time to implement: ________________________

Fix.

Done by: ________________________ Date: ________________

Description:

Time Taken: __________________ Problem tested and signed off by:__________________________ Date:__________