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

Technical Note **LIGO-T950057 - 00-E** 9/29/95

AVS Software Standards Guide

James Kent Blackburn

Original distribution of this document:

Albert Lazzirini, Hiro Yamamoto, Dennis Coyne

Robert Spero, Lisa Sievers

Rolf Bork, David Barker

Andy Kuhnert

CaRT

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

LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY

- LIGO -

CALIFORNIA INSTITUTE OF TECHNOLOGY

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Technical Note **LIGO-T950057 - 00-E** 9/29/95

AVS Software Standards Guide

James Kent Blackburn

Distribution of this draft:

Albert Lazzirini, Hiro Yamamoto, Dennis Coyne

Robert Spero, Lisa Sievers

Rolf Bork, David Barker

Andy Kuhnert

CaRT

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

TABLE OF CONTENTS

1 INTRODUCTION

This technical document is intended to layout guidelines in choices of software languages, software style, software documentation and source code configuration control. The individual LIGO software developer, through the use of common protocols discussed in this document, will produce software that is easy to read and maintain, that is well understood and extensible and that has long lasting value to the project. This document is not meant to be a detailed tutorial or the definitive source of knowledge on the subject. The introductory chapter of the second edition of Numerical Recipes provides the interested reader with other aspects of this subject.

This document will also try to present some of the concepts and current software development tools available under Unix that may assist the software developer in staying close to the less than sharply defined boundaries of software standards. These include the importance that high level languages such as FORTRAN and C have on style, the significance of compiler flags, and the use of poorly documented library functions such as random number generators[1].

Since the aim of this document is to describe software standards for use in AVS, it is only natural that some discussion of software development under the AVS environment be included. Among the points to be covered are a brief overview of what the AVS environment brings to a large software program and what are the interfaces provided by AVS to the traditional high level languages such as FORTRAN and C and how these may be carried forward to include other environments like Matlab and Mathematica. Examples of AVS modules in C and FORTRAN for random number generators are also given in the appendix.

2 SCOPE OF A SOFTWARE STYLE GUIDE

Computer programs are based on source code developed using computer languages such as C and FORTRAN. The reason for using high level computer languages is to make the source code readable by humans, not by machines. This is especially true in a research program such as LIGO where there is little doubt that either through new found knowledge or even as the result of bugs introduced by the complexity of the models, any source code written will at some point need to be revisited either by the author or someone who has taken on the original author's responsibility. To make the task of reading source code as easy and efficient as possible, it is advantageous to introduce and even to demand software coding practices from the very beginning of a project.

But readability is not the only reason for having software style guides. Another even more compelling reason revolves around the issue of standardization. The compilers being used today are not the compilers to be used in the future. Nor are the software and hardware vendors that are currently in place guaranteed to be the primary platforms used even six months from now. By following the guidelines of standards as closely as possible, the tendency to develop backwards compatibility into computer languages will provide forward compatibility for any software developed today.

Still, the use of standards is not always enough. Issues of style can increase portability across current platforms that may be a part of a probable user community of any significant software tool.

---

1. A brief discussion of random number generators is given in Appendix 3.

An example may serve to make the point. The Sun FORTRAN compiler initializes all local variables to zero upon entry into any subroutine or function. In order to do so the Sun FORTRAN compiler must make a table of all variables that appear in the subroutine or function and initialize each variable to the appropriate bit pattern for zero based on the data type of the variable. This happens before actually executing user-specified computations. The advantage of this is that the careless programmer can neglect to set a variable that may be used as a simple counter or as part of a more complex logical algorithm and the program will still work correctly. Even the ANSI standard for FORTRAN does not address this issue. Now here is where the issue of software style becomes important. Most high performance FORTRAN compilers for ultra-fast computers do not take the time to initialize variables the user. These computers were designed to work with extremely large data sets where the time required to initialize these large arrays becomes a major performance issue. These compilers rely on the software developer's understanding, comments and style to make debugging such pitfalls manageable. The software is more easily ported to these platforms if it has comments documenting the use of variables and their initial values.

3 AVS OVERVIEW

Advanced Visual Systems or AVS is a modular software system driven by a 'point and click' mouse interface. The package provides a marvelous set of tools for manipulating multi-dimensional sets of data using very well thought out data structures. AVS modules are functionally equivalent to subroutines in FORTRAN or void functions in C. Modules have a unified communications mechanism among themselves known as AVS ports. Each module will have, depending on its design, a set of input and output ports used to pass data along from one module to the next in a hierarchical scheme known as an AVS flow network. In addition, modules may possess any number of parameters which are controlled by the user through easy to understand graphical widgets such as dials, sliders and text fields. The modules are stored in a graphical palette, much like a library and can be dragged down to the flow networking editor where intercommunications are established using the mouse alone.

AVS supports passing many of the standard data types, such as byte, integer, real and string, between modules. In addition, AVS has field, colormap, geometry, pixelmap, UCD and User-Defined data types for exchanging data. In most cases the most useful of these will be the field and the User-Defined data types. Of these two data types, the field has the advantage of already having modules supplied by AVS for reading and writing the data to file. The field data type is a general representation for an array of data. It also has a self contained description of the data. The field uses either an implicit or explicit mapping of the data elements to coordinates. This provides a very general and very powerful representation for data. The AVS field data type is analogous to a C structure and is defined in the <avs/field.h> header file. Because of this one-to-one relationship between the AVS field and a C structure, the field data type is most easily used from the C language by a software developer fluent in C. However, AVS does provide a set of subroutines to access and manipulate any desired elements of the field data type from FORTRAN. Using the FORTRAN calls to access the field adds an extra burden on programmers revisiting the source code since they would now need to understand the purpose of the calls to the AVS FORTRAN library. As is always the case when using non standard library routines, the software developer should fully document through comments the purpose of all calls to the library routine.

The AVS module represents one procedural unit of computation. Each module takes up a process ID slot in the Unix kernel[1]. Too many modules can fill up the available process slots in the kernel and cause the flow execution of the AVS network to fail. To avoid this, modules should not be so minimal in their function as to require many modules to actually accomplish something useful. For example, it would not be very practical to have a module for subtracting two integers. On the other hand, if too much functionality is placed into a single module, such as having the entire End-To-End Model as a single AVS module, then it would have too specific utility to be used in other applications and any variations of the Model would require re-construction. So modules should be unique enough in there functionality to be useful to multiple applications.

Software developers can easily develop specialized AVS modules using the C or FORTRAN languages. This basically involves generating AVS "wrappers" around C functions or FORTRAN subroutines. The "wrapper" is responsible for setting the AVS module name, creating input and output ports for the module, creating parameters for the module, setting the computation function for the module, and performing any additional initialization that is required. AVS has its own error handling mechanism which includes an error logging facility. It should be used instead of print statements in the body of the source code. Debugging of modules is also supported using the **avs_dbx** shell script On a Sun workstation using Solaris, the actual command to be used for debugging is **avs_dbx -debug debugger *module*** from the directory where ***module*** is located. For more information on AVS and developing modules in the AVS environment see the AVS documentation[2].

Any character strings of text printed to standard out by an AVS module will appear in the shell that was used to start up AVS. To avoid having to much information in that shell's window, it is recommended that a verbose parameter be provided to the AVS module that is an integer represented by a slider widget that ranges from 0 for no output printed to some maximum positive integer for print everything. This could prove to be a useful debugging feature in modules as they are propagated from the developer to the user.

Dynamic memory management is an important aspect of any AVS module. This is because the AVS module exists as a Unix process. Each time the module is run it should free up any memory that was previously allocated and then allocate the required memory again. This will prevent memory leaks from occurring when the module is repeatedly run with new parameters. AVS provides routines for dynamic memory allocation and these should always be used instead of the Unix equivalents.

4 SOFTWARE STYLE

This section addresses issues of software style and is intended to provide some guidelines to the various topics that bring 'added value' to source code. Among the key topics are readability, portability, structured programming, the use of popular high level programming languages such as C, FORTRAN and C++ in the AVS environment, along with a discussion of AVS software develop-

1. It is possible to place multiple modules in the same process. See page 4-27 of AVS Developer's Guide.
2. Recommended in the following order: AVS Tutorial Guide, AVS User's Guide, AVS Application Guide, AVS Developer's Guide, AVS Module Reference

ment. The software developer should consider these topics when writing his or her own software so that others may also benefit from that software.

4.1. Software Readability

There are many common sense approaches to software readability. The most highly promoted of these is the use of comments. Using comments does not mean to 'spray' one's source code with non-compiled words. Comments should be viewed in a systematic way. There should be a header to the code where a set of descriptors like **program name, author, creation date, modification history** and **purpose** are listed[1]. These can be and often are standardized across a project. Comments should also be used to describe formulae which should include references. Any use of library routines should be documented clearly. Understanding why a non-standard library call is used is one of the most time consuming aspects of revisiting source code. The same is true of subroutines and functions. Comments should also be clear and concise. Any lengthy explanation should be made in some form of documentation[2] and referenced.

A second area where readability is important is in expressing complex, lengthy formulae. Instead of trying to reproduce an expression as closely as possible to its analytic form, it is best to program sub-expressions and combine them in steps using several compiler statements instead of using continuation statements in an attempt to represent the expression in its original form. This can have the added advantage of improving performance when sub-expressions appear repeatedly and can improve performance. Breaking up expressions if also important for FORTRAN source code where the ANSI standards restrict the statement length to 72 characters[3].

Avoid being tricky with programming statements. There is almost an attitude among C programmers that as much as possible should be packed into the fewest number of characters possible. Try to avoid this temptation; it may win first place in the most esoteric one line C program contest, but it won't make friends among colleagues trying to find the bug you left in the software. It is also very helpful to give variables names that are somewhat self evident. However, care must be used to avoid violating ANSI standards for variables. All C preprocessors implement a set of macros used to control compilation. These macros vary from one compiler to another but always begin with an underscore. In addition, C-FORTRAN code binding in Unix usually requires adding an underscore on globals such as function names.[4] So use underscores as the first character in C with care. ANSI FORTRAN 77 doesn't allow more than 6 characters in a variable name which makes code less readable. Most Unix FORTRAN compilers allow more than 6 characters in names so use this extension unless you are developing code that may have to run on many different platforms where the longer variable names may cause more difficulty than is gained through the readability of these longer names. Also most Unix FORTRAN compilers allow the use of lower case characters. This tends to improve the readability of code as well, though it is also a violation of the ANSI FORTRAN 77 standard. Case of characters can easily be changed if needed, so do not give different FORTRAN variables the same names while differing only by the case.

---

1. For a complete list of header keywords, see Appendix 4.
2. See Section 5 of this Documentation
3. AVS FORTRAN wrappers adhere to this standard implying that AVS runs on platforms requiring it.
4. See page G-50 of the AVS Developer's Guide for additional details.

4.2. Software Portability

The importance of software portability is to make software written by one person on one computer platform with its specific compilers usable by others on other systems. Portability at a future date is also an issue for software. The operating systems and compilers will change with time. AVS may not be the environment LIGO uses 10 years from now. By developing portable algorithms at the heart of AVS modules, it will be possible to migrate to the AVS successor with minimal effort. The primary tool for maintaining software portability is to follow the ANSI standards for computer languages. This will typically mean ANSI C and ANSI FORTRAN 77[1]. There are several compiler flags and tools available for determining ANSI compliance and portability. For Sun's FORTRAN, try the **-ansi** compiler flag. There is also a public domain utility which is highly recommended called **ftnchek**[2]. It checks much more than ANSI compliance and is very useful for finding programming errors. For C, the standard tool for checking portability of code is **lint**. It also is useful for finding errors. These tools have Unix man pages and should be reviewed by all software developers. These tools are only aids, the programmer ultimately is responsible during its development for the portability of the software. The software developer must fully understand the flow of the code, which will be discussed further in the next section.

4.3. Structured Programming

As a program executes, the ordering of the statements unfolds through the use of control statements like *if-else*, *do-while*, *do-continue*, etc. Structured programming is the use of control statements in such a way as to make the flow of a program visually apparent to the reader of the source code for the program. Well structured code is also easier to develop and to debug. It is not possible to design structure into programs that use goto statements and label statements. Code written in this style is routinely referred to as "spaghetti-code" and should be avoided unless absolutely required in certain very narrow and specific instances. Standard FORTRAN 77 has minimal support for structures similar to the while loops of other languages. However, with some creative thought on the placement of *continue* statements and *if* statements preceding a tame *goto*, it is possible to implement many of these structures.[3] Another important aspect of structured programming is to keep things modular. Design the software to flow into and out of blocks of code, not lines of code. Additionally, don't overload the functionality of subroutines and functions.

4.4. Programming Languages in AVS

The developer of AVS software modules has the freedom to choose between C, FORTRAN, C++ and other highly specialized software environments like Mathematica and Matlab. The C language provides the most natural constructs for working in AVS, but the others have advantages that may factor into the decision to use one over the other.

---

1. FORTRAN 90 compilers are not widely available.
2. See Kent Blackburn for the source distribution.
3. See Chapter 1 of Numerical Recipes (FORTRAN version) for examples.

4.4.1. Programming with C

AVS module development uses constructs such as dynamic memory allocation and data structures that are very familiar to the C programmer. Because of the similarities that exist between these two high level environments, the C programmer will experience the fastest learning curve when developing AVS modules with C. Additionally, many of the data types used in AVS can be directly manipulated from C since these AVS data types have a one-to-one relationship with C structures. However, C compilers have yet to reach the level of optimization that is currently available with FORTRAN unless the programmer writes her C code in the style of FORTRAN. In addition, the scientific analysis libraries available to C programmers are not as complete, nor as well tested as the scientific libraries for FORTRAN programmers. Still the C language is more structured and modular in its constructs. C code tends to be more portable as well, primarily because the C programmer is less likely to need to resort to compiler specific calls to support the needed algorithm.

4.4.2. Programming with FORTRAN

AVS includes a complete set of tools for developing modules in FORTRAN. This an extremely useful and important feature of AVS. FORTRAN compilers tend to be several times more efficient in numerically intensive software than other languages. There is also an extensive set of numerical libraries available to the FORTRAN programmer which have been thoroughly tested. Support for FORTRAN is also important to the End-To-End model since there is a significant amount of software that has already been developed which will be ported over to AVS modules. However, the handling of data and the memory management and the inter-module communications of AVS are addressed using AVS library routines. The naming convention for these libraries provide little more than a clue to the purpose of the call. So to make the intent of the code more clear, software developers should clearly comment the calls to AVS routines.

4.4.3. Programming with C++

Object oriented languages add a very powerful collection of concepts to high level languages.[1] Chief among these are the linking of functionality to the data, truly reusable code and polymorphism. AVS does not provide a clean interface to C++, instead module wrappers for the C language must be slightly modified to be compatible with a C++ compiler.[2] A crude AVS module, implemented as a C++ class is demonstrated in the AVS examples.[3] If you were *really* going to use C++ within a module, you would probably use the AVS functions as they are and develop the internal code of your modules using your own class libraries. This does not provide for a very robust implementation of C++ support and is therefore only recommended for modules where concepts of polymorphisms, inheritance, operator overloading and other C++ constructs will provide a major time savings in future source code development.

---

1. See Appendix 2
2. See section 8 of AVS 5 Update for complete details.
3. See the file $AVS_PATH/examples/cpp_example.cpp

4.4.4. Programming with Matlab and Mathematica

Both Matlab and Mathematica provide interprocess communications. For the AVS module developer, the direction of communications which will appear most frequently is to have the AVS module request that some analysis be carried out by the Matlab or Mathematica computational engine. But AVS does not support direct communications to the Matlab and Mathematica environments. To accomplish this, either C or FORTRAN must be used to cross the bridge. In the case of Matlab, the AVS module developer has the choice of interfacing with the Engine Library[1] through either C or FORTRAN. Using MathLink[2] which is distributed as part of Mathematica, the AVS module must communicate through MathLink C library functions. In all cases, deciding to use the Matlab or Mathematica computational engine should be based on a need to provide a solution quickly, because these tools require licenses and may not be available to others which means that your AVS modules may not be available to others. In addition, both Matlab and Mathematica may carry with them a performance hit that could be lessened by a compiled version of the code written in C or FORTRAN. Still having said all this, these are potentially useful interfaces to have available to implement from AVS modules.

4.5. Developing AVS modules

AVS modules are developed using the GUI driven **Module Generator** module found under **Data Output** in the **AVS Module Library** of the **AVS Network Editor**. Pulling down this module to the network flow palette produces a set of options for generating the C or FORTRAN wrappers for routines to be provided by the developer.[3] These wrappers are even complete with comment headers specifying the module name, author, and date created. There are four[4] separate sections within the wrapper. Each section is marked by a pair of comments that look as the following in C

```
/* ----> START OF USER-SUPPLIED CODE SECTION#...*/
/* <---- END OF USER-SUPPLIED CODE SECTION #...*/
```

and look as the following in FORTRAN.

```
C ----> START OF USER-SUPPLIED CODE SECTION#...
C <---- END OF USER-SUPPLIED CODE SECTION #...
```

It is crucial that developers place their lines of code between these comments. If not, when a modification to the AVS interface occurs, any lines of code outside the comments are lost! The types of interface changes that would cause this are changing the number of parameters, input ports or output ports. Also changes to the data types used by any of these. And of course, any time the user chooses the **Write Source** button for the **Module Generator**.

There are occasions when the developer of AVS modules will need to extend the functionality provided by the standard AVS interface. Two occasions for this are very likely. The ability to

1. See the MATLAB External Interface Guide for details.
2. See the Wolfram Research technical report, The MathLink Reference Guide for details.
3. See Section 2 of the AVS Applications Guide for a simple example or writing an AVS module.
4. FORTRAN wrappers actually have only three.

write code to address ports in the USER-SUPPLIED CODE SECTIONS is very poorly supported in AVS. To address these ports, it is necessary to modify the code written by the **Module Generator**, allowing for unique identifiers. If this is the case, comments placed around the lines of code that have been modified will act as a marker to others reading this code that this section must be reconstruct that by hand if the **Module Generator** is used to modify the AVS interface such as the parameters, ports or data types. In addition, make a backup copy of the changes in comment form in the closest standard USER-SUPPLIED CODE SECTION with detailed instructions on where the modifications should be recreated in the event that they are accidently lost through the use of the **Module Generator**. Any comments that surround these modifications must have a standard form to be useful in an automation script. For C code use the following

/* ====> START OF LIGO USER MODIFICATION TO AVS CODE SECTION */

/* <==== END OF LIGO USER MODIFICATION TO AVS CODE SECTION */

and in FORTRAN use these.

C ====> START OF LIGO USER MODIFICATION TO AVS CODE SECTION

C <==== END OF LIGO USER MODIFICATION TO AVS CODE SECTION

Be sure to implement these comments exactly. A future script can be developed to automate the process of making the needed modifications. Additional comments should also appear within these comments to expand the purpose of the modifications to the code originally written by the **Module Generator**. This is important since AVS may actually change the interface to the ports and parameters in a future version and the exact line of code used to make the modification would have no value, only their purpose could be used to make the needed changes.

5 DOCUMENTATION

The generic purpose of documentation is to furnish information and present proof of principle. This applies without exception to software development. Through comment statements in the source code, the proof of principle is documented for software developers. Through User Guides, on-line help such as the AVS module help standard and Unix man pages, information is furnished to the end users.

5.1. Source 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 header section in the source code built out of comment statements that present the summary of the software. This should include, but not 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. All of these header comments will 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 place within the body of the code. These comments are for explaining the flow of the code, describing the purpose of subroutine and function calls, and describing 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.

5.2. AVS On-line Module Help

AVS has a standard interface for providing help on modules. This help is GUI based and is initiated by clicking on the small square button on the right side of each AVS module with the right mouse button. This brings up a panel that lists all parameters and ports for the module by their names and is color coded to their data types. At the bottom of this panel are a set of buttons, the first of which is labeled **Show Module Documentation**, and is used to get help in the style of a Unix man page. This panel also has several buttons used to control the behavior of the module that are not relevant to the present discussion. When the **Show Module Documentation** button is selected, a scrollable window appears which provides on-line help consisting of the module name, summary information, a general description, list of parameters, input and output ports, references to other AVS modules and any limiting features for the module. The developer of AVS modules must generate this on-line help for each module. This is done very simply by clicking on the **Write Man Page** button for the **Module Generator**. This will produce a file with the module's name followed by the extension ***.txt** which must then be edited to fill in the details. This is a simple text file so care should be taken to limit lines to 80 characters so as not to cause line wrap problems in the AVS help window. The value of this on-line help in furnishing information about the purpose of an AVS module is clear and its creation should not be neglected.To see the help page, the environment variable **AVS_HELP_PATH must** be set to the location of the ***.txt** help file.

5.3. User's Guides

Software documentation should also include User's Guides which provide both information about the use of the software and technical details justifying the formulation of the software. User's Guides should be more descriptive than on-line help pages about the use of the software by including details and examples. The User's Guide is the ideal place to present the models, formulae and numerical techniques that went into the software. It should also be the place to discuss any limitations and directions for future enhancements. This is also the type of documentation that is most likely to be found in paper form. As such it would naturally fall under the requirements of document control standards. Since it is useful to access such documents from a computer, they must be in a format that is available from Web browsers.

Using the Webmaker[1] program described under the LIGO Web pages, it is possible to convert Framemaker files into HTML documents that are viewable over the Web. It is very important to use the standard LIGO templates with Webmaker to guarantee proper interpretations of the Framemaker files. Another translator with is available takes LaTeX files and converts them to HTML. There are also HTML editors such as HoTMetaL[2] available for writing Web pages.

1. The URL is http://docuserv.ligo.caltech.edu/docuserv/home/convert_1.html (link from LIGO homepage).
2. To use HoTMetaL use the command sqhmpro from your LIGO Solaris Workstations.

Clearly, the route to take is to have User's Guides written as Framemaker files making printed versions available for document control purposes and then translate them to HTML format for use with Web browsers. Web browsers often have limited resolution and bandwidth when view large documents or documents with detailed diagrams so postscript versions of HTML documents should also be distributed by FTP from the HTML source and made available for downloading and printing.

5.4. Unix Man Pages

Template Unix man pages for modules can be generated from within AVS. This has the advantage of allowing Unix users to study the function of an AVS module without being in the AVS environment. AVS on-line help pages are pre-formatted into text files. This format is not suitable for Unix man pages. However, the AVS **Module Generator** can produce troff formatted files which can easily be processed through nroff to produce true Unix man pages. To do this the environment variable **AVS_MG_TROFF** must be set before starting AVS. This is accomplished by using setenv AVS_MG_TROFF from the shell which will be used to start AVS. Then when the **Write Man Page** button for the **Module Generator** is selected, the help file will be generated in the troff format. At the top of this file in the comments are instructions for turning this file into a Unix man page. The instructions are something like

nroff -ms -u -rM62 -Tlp /home/kent/avs_dir/mymodule.txt > mymodule.1

for a help file named mymodule.txt created by the **Module Generator**. This will create a file named mymodule.1 or whatever is appropriate for your module's name. Before the Unix man command can access this man page, it must be placed in the Unix man directory tree structure. This is a directory tree located at a commonly accessible place in the Unix file system[1] with the top level name of man. Within this directory is a subdirectory named man1 which should become the home for the mymodule.1 man page and any others that share this **MANPATH**. To find this man page, the path to the top level man page must be specified in one of two ways[2], either by specifying the path using the **MANPATH** environment variable, or by using the -M man option followed by the path. To facilitate the use of man pages, a LIGO wide man directory tree structure needs to be implemented on the LIGO file servers. This should have expanded functionality over what is currently available with the ligobin and ligoman directories[3]. For example, making the source code management software and the shared module library requirements of AVS part of the directory tree, allows the LIGO group to access and share a common version of software.

6 SOURCE CODE MANAGEMENT

Source code management or version control as it is often called is a very important aspect of large software projects. There are several public and commercial systems available for automating many tasks associated with coordinating a team of software developers. The tasks included by most of these systems are maintaining all versions of a program in a recoverable form, prevention

1. See Appendix 5 for the directory tree structure.
2. See the man page for man for details.
3. Found under /home/ufs2 on the LIGO servers.