LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY

- LIGO -

CALIFORNIA INSTITUTE OF TECHNOLOGY

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Document Type **LIGO-T980067-00 - E** July. 1998

Extending E2E Model - The How To Guide to Coding Modules

Biplab Bhawal, Matt Evans, Edward Maros,

Malik Rahman and Hiro Yamamoto

Distribution of this draft:

xyz

This is an internal working note
of the LIGO Project.

California Institute of Technology
LIGO Project - MS 51-33
Pasadena CA 91125
Phone (626) 395-2129
Fax (626) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project - MS 20B-145
Cambridge, MA 01239
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

WWW: http://www.ligo.caltech.edu/

1 ABSTRACT

This document is intended to show a user of the End to End simulation package how to add new functionality by creating new modules. The is not a comprehensive document explaining how to write modules in general, but is a tutorial using simple examples. Examples are given using C++, C, or Fortran.

2 KEYWORDS

Adlib, C++, C Plus Plus, Fortran, module, creating

3 OVERVIEW

End to End model is a program developed for the time domain simulation of LIGO experiment. The simulation program is written in C++, and is designed so that new capabilities, like special hardware behavior or non standard noise, can be easily added by writing a set of codes, called a primitive module. This is achieved by hiding the book keeping part from the user so that the user can concentrate on writing codes dealing with the subject of interest.

In the first version of the simulation package, all of the primitives were compiled into a single program. To add a new primitive required creating a new C++ module and then rebuilding the whole program. This method has several disadvantages.

First, the end user needs to know C++ programming. This limits the number of people who would be able to contribute to add new physics by building new modules.

Second, it requires special version of the program to be compiled for each user who wants to have a new set of modules. Not only is this a waste of disk space, it also makes maintenance very difficult.

Third, when you want to use modules written by other people, you have to rebuild your program including new source codes.

For these reasons, a new approach was developed. The program now focuses on the running of the simulation, i.e., it contains codes to maintain the time domain simulation and minimum set of primitive modules. When you run the program, primitive modules are dynamically loaded into the system, i.e., new functionality built in those modules loaded are added to the program. Since they are dynamically loaded, there are following advantages:

- languages other than C++ can be used easily
- one program can be used by all users
- one can include capability written by other people very easily
- one can define new primitives specialized to their problem, possibly overriding existing primitives.

This document focuses on the programming aspects of creating a new primitive module. The document, *Getting Started with End-to-End model LIGO-T980051-00-E,* is the tutorial of the End to End model, and it contains other references for in depth information. The documents, *Overview of*

End-to-End model LIGO-T970193-00-E and Organization of End-to-End model LIGO-T970194-00-E, will give all information how to write modules in C++. In order to support C and Fortran in the level as is explained in this document, different wrappers, or support files, are needed. Please contact one of the authors about the availability of wrappers for the modules you are interested in.

For now, the supported development environment is the latest version of egcs. This contains C++, C and Fortran. If you prefer to use other environment for the module development, use it until you complete the testing of your code, and do the final build using egcs. C++ compiler of egcs is very close to ANSI standard, and the fortran compiler is based on FORTRAN 77 with reasonable extensions. If your code conforms to the standard, the final build will not be a problem.

4 CONVENTIONS USED IN THIS DOCUMENT

All of the source code for the samples is located in the appendix labeled "SOURCE CODE SAMPLES." Please note that the c++ sample is the only one needing a programmer defined header file. In the codes included in this document, those parts are shown in bold face which need to be modified when you write a new module.

5 FEATURE HISTORY

Initial Release

6 PROGRAM STRUCTURE

The primitive module explained in this document is the kind which accepts several real values and calculate one real value, i.e., like a c function

```
double foo( double d1, double d2 ) { double sum; sum = d1 + d2; return sum; };
```

When you create a new module of this kind to add a new functionality to the End to End model, only extra work you have to do is to define the interface of your module, i.e.,

- specify inputs by the number of inputs, their names and default values - values used when nothing is connected.
- define parameters of the module, like a reflectance of a mirror, which are constant during the simulation, by names, types and default values - values used when nothing is specified by the end user.

After you write this module, put it in the simulation link (if you have not read the tutorial or the overview document, please do so now). When you run the simulation, your module is called repeatedly at each time step, t_0, $t_0+\Delta t$, $t_0+2\Delta t$... with proper inputs from modules preceding to your module, and the output of your module is passed into other modules following your module.

7 GETTING STARTED

The simplest way to get started is to create a directory to store the source code and various support files needed for the module. Copy all the files from the directory e2e/Example/<language> where language is either c, c++, or fortran.

Following standard conventions, c source code should have the extension ".c", c++ source code should have the extension ".cc", and fortran should have the extension of ".f".

8 MAKEFILE

Each example comes with a Makefile. This file contains all instructions to build a program, which files are to be compiled, how to be compiled and how to be linked.

This is the first file that should be modified so that your codes are to be included. There are two sections that need to be modified. The first section has lines of the form MODULES += <filename without extension>. Have one line of this form for each of the modules you are creating. In the example that is supplied in the "SOURCE CODE" appendix, there is a module being created, flimiter. It is important not to add the file extension.

The second section which requires periodic changing is the revision numbers. There are three variables that are suppled. The first, "RELEASE_MAJOR" indicates a major change in the shared object which makes is incompatible with previous versions. The second, "RELEASE_MINOR" indicates a minor change which should not affect existing models. Lastly is "RELEASE_PATCH". This is used to indicate bug fixes.

9 A C++ EXAMPLE

This interface allows the greatest flexibility. A programmer can derive from any of the subclasses of modules. This flexibility comes at the cost of knowing how to program in C++. It is not the purpose of this document to teach C++ to the programmer, but instead to explain how the C++ interface can be used.

The example to be used is a variation on the limiter primitive. Like the limiter primitive, elimiter is derived from real_function and limits the output. It differs from the limiter primitive by having the high and low ranges being parameters instead of additional input ports.

The first stage in creating a module is the MI_SO_INIT(). The purpose of this function is to create an entry point to the shared object. This entry point does the basic initialization that is required by the Adlib engine. Each module needs a unique key (elimiter in this sample). MI_KEY is the define of the key. This key needs to be defined before the inclusion of the mi.h (module interface) file. MI_SO_INIT() should appear just above the class constructor.

There are four methods that a module should define. The first two are the constructor and destructor methods. The constructor receives name_arg and parent_arg as its arguments. These should be

passed to the base class. Data members should be initialized to their default values at this step (warned, upper, and lower in this example).

In the body of the method should be the setting of ports and auxiliary data members (settings). Set_default_input() receives the port offset (starting with 0), of the port and the default value it should receive (the value used when nothing is connected to the port). In the elimiter example, port 0 has a default value of 0.0.

Add_auxiliary() receives a data_ref and the name of the setting. Data_ref() receives the address of the data and the data type. There are two settings in the elimiter example. The first is the upper bound. It has a data type of Type_Real, address of upper, and a name of "upper." The second is the lower bound. It also has a data type of Type_Real, address of lower and a name of "lower."

The destructor function should be deallocate any dynamically allocated resource created by the programmer in the constructor function. In the elimiter example there is no such data, therefore, the destructor function is empty.

The third method that needs to be constructed is new_type(). This method is how the Adlib engine creates new instances of the object. It receives two parameters (name_arg and parent_arg) and returns a pointer to a module (the base class of all modules). The code simply new's an instance of the derived class passing name_arg and parent_arg and returns the newed instance.

The last method that needs to be constructed is action(). This is a void function that does a single iteration of calculations. Using the current inputs and settings, all outputs are calculated and made accessible.

10 A C EXAMPLE

The C interface is currently limited to real functions. The example used for the C interface mimics the C++ interface example. It has one input, one output and two settings (upper bound and lower bound).

The form of the C interface is:

```
#include "rfframe.h"
#define MI_KEY <KEY>

static double action(double* Parms);
static enum {
  <AUX DEFINITIONS>
  AUX_END,
  <INPUT PORTS> = AUX_END,
  IN_END
};

static DATA parms[IN_END] =
{
  /* Set up all of the auxiliary data */
  <AUX SETTINGS>
  /* Set up the input ports */
  <PORT SETTINGS>
};
```

```
static double
action(parms)
     double* parms;
{
  <Set Local Variables>

  <Do Calculations>
  <Return value>
}

RFFRAME_ENTRY
```

<KEY> is the unique key to identify this type of module (climiter in this sample).

The "static enum" definitions are used to reference the setting parameters and the input ports. The settings values appear first (<AUX FEFINITIONS>) and are just listed one after the other (AUX_UPPER and AUX_LOWER in this sample). The last setting must be followed by AUX_END. After the settings have been declared, the input ports are to be declared. The first input port should have the form <port> = AUX_END where <port> is the name of the enumerator describing the first port (IN_0 in this sample). The last input port must be followed by IN_END.

The "static Data parms[IN_END]" provides the default values for each of the settings and ports. The form should be a character string followed by a double.

The function must be named action and receive as its only argument an array of doubles. It returns the value that should appear on the output. For the sake of speed, it may be good to store values from the argument in local variables.

The "RFFRAME_ENTRY" line at the end of the file is very critical as it does glue everything together.

11 A FORTRAN EXAMPLE

Currently, the only supported fortran compiler is the one that comes with egcs. Please have your system administrator read the appendix entitled, "COMPILING FORTRAN WITH EGCS" for special installation instructions.

The fortran sample is flimiter and it mimics the c++ and c sample programs. The fortran interface requires the defining of two functions. The first function is INIT(). It takes five parameters, ELEMENTS, AUX_CNT, CHARSIZE, DEFAULTS, NAMES. The first three of these (ELEMENTS, AUX_CNT, and CHARSIZE) are of type INTEGER. ELEMENTS is the total number of settings and input ports, AUX_CNT is the number of settings, and CHARSIZE is the length of the longest character string. DEFAULTS is an array of REAL*8 which holds the default values for the settings and the input ports. NAMES is an array of CHARACTER*VARLEN which holds the names for the settings and the input ports. It is recommended to use a block of PARAMETERs to define constants. Two that are highly recommended are VARLEN which defines the longest string name used and DIM_END which indicates the number of settings and input ports. Other PARAMETERs that are recommended are those used to indicate positions of settings and input ports

(AUX_UPPER, AUX_LOWER, and IN_0 in this sample). By defining these, the programmer increases the ease with which to add additional settings and input ports later.

The INIT() function serves two purposes. The first is to tell the caller how much space is required. This is achieved by the caller passing a value in "ELEMENTS" that differs from "DIM_END." When this is the case, the function should return correct values for ELEMENTS, AUX_CNT and CHARSIZE.

The second use of the INIT() function is to initialize DEFAULTS and NAMES. With ELEMENTS equal to DIM_END, there should be enough space allocated.

The second function that the programmer must create is the ACTION() function. This function takes an array of REAL*8 as its only argument and returns the value of the output port. The array of REAL*8 contains the values of the settings and the input ports.To help with efficiently, it may be helpful to store some or all of the values from this array into local variables and then use the local variables in all calculations. A programmer should never change the values of the array.

APPENDIX 1 MAKEFILE - SAMPLE

```
#=====================================================================
# DO NOT CHANGE
# General Stuff
#=====================================================================

MAKEFILE_ID="$Source: /home/e2e/Software/cvsroot/e2e/Example/fortran/Make-
file,v $"

-include ../Makefile.cf
ifneq (,${E2E_MAKEFILE_CFG})
-include ${E2E_MAKEFILE_CFG}
endif

#*********************************************************************
# Language Specification - Change -
# Specify the language being used (Fortran in this case)
#  Valid values are: C++, C, Fortran
#*********************************************************************

MODULE_LANGUAGE=Fortran

#=====================================================================
# DO NOT CHANGE
# Include special things needed to build modules
#=====================================================================

include $(CONFIG)MMakefile.cf

#*********************************************************************
# Module Specification - Change -
# Specify the names of the modules to create (flimiter in this case)
#*********************************************************************
```

```
MODULES  =
MODULES += flimiter

#***************************************************************************
# Release - CHANGE -
#     When a new release of your modules is to be made, change the
#     numbers here.
#     MAJOR usually indicates a change that prevents it from working
#           with prior versions.
#     MINOR usually indicates a change that adds or changes
#           functionality without affecting prior versions.
#     PATCH if for small changes that corrects behavior.
#***************************************************************************

PROJECT= FortranExample
RELEASE_MAJOR= 1
RELEASE_MINOR= 0
RELEASE_PATCH= 0
RELEASE    = $(PROJECT)_$(RELEASE_MAJOR)_$(RELEASE_MINOR)_$(RELEASE_PATCH)

#===========================================================================
# DO NOT CHANGE
# Include special things needed to build modules a second time to
# expand variables and get default rules.
#===========================================================================

include $(CONFIG)MMakefile.cf
```

APPENDIX 2 SOURCE CODE SAMPLES

A2.1. A C++ Sample - elimiter.cc

```
/*-------------------------------==---------------------------------*\

File: elimiter.cc

This is a simple sample program of how to create a limiter. It
works just like the system limiter, but instead of having the
upper and lower bounds being inputs, they are settings.

\*-------------------------------==---------------------------------*/

#include "elimiter.h"

#defineMI_KEY elimiter

#include "mi.h"

MI_SO_INIT()
```

```
/*-------------------------------==---------------------------------*\
                                elimiter
\*-------------------------------==---------------------------------*/

elimiter::
elimiter(const string& name_arg, const module* parent_arg)
  : real_function(name_arg, parent_arg, "elimiter", 1),
    warned(false),
    upper(0.0),
    lower(0.0)
{
  set_default_input(0, 0.0);

  add_auxiliary(data_ref(&upper, data_ref::Type_Real), "upper");
  add_auxiliary(data_ref(&lower, data_ref::Type_Real), "lower");
}

elimiter::~elimiter()
{
}

module* elimiter::
new_type(const string& name_arg, const module* parent_arg) const
{
  return new elimiter(name_arg, parent_arg);
}

void elimiter::action()
{
  output = in(0);
  if (upper >= lower)// Sanity check
  {
    //-------------------------------------------------------------------
    // The range is valid. See if the output needs to be cropped
    //-------------------------------------------------------------------
    if(output > upper) output = upper;
    if(output < lower) output = lower;
  }
  else if (!warned)
  {
    //-------------------------------------------------------------------
    // Give a single warning.
    //-------------------------------------------------------------------
    mess(WARNING) << get_name()
  << "lower bound (" << lower
  << ") exceeds upper bound (" << upper
  << ") output will not be modified"
  << endl;
    warned = true;
  }
}
```

A2.2.A C++ Sample - elimiter.h

```
/*-------------------------------==-------------------------------*\

File: elimiter.h

\*-------------------------------==-------------------------------*/

#ifndef __ELIMITER_H__
#define __ELIMITER_H__

#include "real_function.h"

/*-------------------------------==-------------------------------*\

The "elimiter" models a circuit with rails.

\*-------------------------------==-------------------------------*/

class elimiter : public real_function
{
 public:
  elimiter(const string& name_arg = "", const module* parent_arg = NULL);
  ~elimiter();

  module* new_type(const string& name_arg, const module* parent_arg) const;

  void action();

 private:
  boolwarned;
  adlib_realupper;
  adlib_reallower;
};

#endif// END:  #ifndef __ELIMITER_H__
```

A2.3.A C Sample - climiter.c

```
#include <stdio.h>

#include "rfframe.h"

#define MI_KEY climiter

static doubleaction(double* Parms);
static enum {
  AUX_UPPER,
  AUX_LOWER,
  AUX_END,
  IN_0 = AUX_END,
  IN_END
};
```

```
static DATA parms[IN_END] =
{
  /* Set up all of the auxiliary data */
  {"upper", 0.0},
  {"lower", 0.0},
  /* Set up the input ports */
  {"0", 0.0},
};

static double
action(parms)
     double* parms;
{
  doubleoutput = parms[IN_0];
  doubleupper = parms[AUX_UPPER];
  doublelower = parms[AUX_LOWER];

  if (upper >= lower)
  {
    if (output > upper) output = upper;
    if (output < lower) output = lower;
  }
  else
  {
    fprintf(stderr, "lower bound(%g) exceeds upper bound(%g)\n", lower, upper);
  }

  return output;
}


RFFRAME_ENTRY
```

A2.4.A Fortran Sample - flimiter.f

```
      INTEGER FUNCTION INIT(ELEMENTS, AUX_CNT, CHARSIZE,
     $     DEFAULTS, NAMES)
*     ------------------------------------------------------------------
*     common block
*     ------------------------------------------------------------------

      INTEGER AUX_UPPER, AUX_LOWER
      INTEGER IN_0
      INTEGER DIM_END, VARLEN

      PARAMETER (AUX_UPPER = 1)
      PARAMETER (AUX_LOWER = AUX_UPPER + 1)
      PARAMETER (IN_0 = AUX_LOWER + 1)
      PARAMETER (DIM_END = IN_0)
      PARAMETER (VARLEN = 5)

*     ------------------------------------------------------------------
```

```
*     Parameters
*     ------------------------------------------------------------------

      INTEGER ELEMENTS
      INTEGER AUX_CNT
      INTEGER CHARSIZE
      REAL*8 DEFAULTS(*)
      CHARACTER*5 NAMES(*)

*     ==================================================================
*     Initialize the values for things that will need to get passed.
*     ==================================================================

      INIT = DIM_END
      IF (ELEMENTS .EQ. DIM_END) THEN
         NAMES(AUX_UPPER) = 'upper'
         DEFAULTS(AUX_UPPER) = 1.0

         NAMES(AUX_LOWER) = 'lower'
         DEFAULTS(AUX_LOWER) = 2.0

         NAMES(IN_0) = '0'
         DEFAULTS(IN_0) = 3.0
      ELSE
         ELEMENTS = DIM_END
         AUX_CNT = AUX_LOWER
         CHARSIZE = VARLEN
      END IF
      END

*     ==================================================================
*     This is where the real work is being done. This is the function that
*     will be executed.
*     ==================================================================

      REAL*8 FUNCTION ACTION(PARMS)

*     ------------------------------------------------------------------
*     common block
*     ------------------------------------------------------------------

      INTEGER AUX_UPPER, AUX_LOWER
      INTEGER IN_0

      PARAMETER (AUX_UPPER = 1)
      PARAMETER (AUX_LOWER = AUX_UPPER + 1)
      PARAMETER (IN_0 = AUX_LOWER + 1)

*     ------------------------------------------------------------------
*     parameters
*     ------------------------------------------------------------------

      REAL*8 PARMS(*)
```

```
*     ------------------------------------------------------------------
*     function implimentation
*     ------------------------------------------------------------------

      REAL*8 UPPER, LOWER

      UPPER = PARMS(AUX_UPPER)
      LOWER = PARMS(AUX_LOWER)
      ACTION = PARMS(IN_0)
      IF (UPPER .GE. LOWER) THEN
         IF (ACTION .GT. UPPER) THEN
            ACTION = UPPER
         END IF
         IF (ACTION .LT. LOWER) THEN
            ACTION = LOWER
         END IF
      ELSE
         PRINT *, 'WARNING: lower bound (', LOWER,
     $          ') exceeds upper bound (, UPPER, ')'
      END IF
      END
```

APPENDIX 3 COMPILING FORTRAN WITH EGCS

Change to the top most directory of the egcs distribution. This will be referred to as EGCS for the remainder of this appendix.

Run the configure script being sure to include the "--enable-shared" option.

Run "make" to build all of the system.

Change directory to EGCS/<system>/libf2c (for example EGCS/sparc-sun-solaris2.6/libf2c).

Run "make mostlyclean"

Change directory to EGCS

Add -fpic option to the CFLAGS and CXXFLAGS variables in Makefile

Run "make" to rebuild the f2c library.

Run "make install" to install the package for others to use.