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

Technical Note	**LIGO-T030154- 00- E**	7/31/03

Creating an e2e Primitive

Jeff Jauregui (2003 SURF Student)
Mentor: Hiro Yamamoto

This is an internal working note
of the LIGO Project.

California Institute of Technology	**Massachusetts Institute of Technology**
LIGO Project – MS 51-33	**LIGO Project – MS 20B-145**
Pasadena CA 91125	**Cambridge, MA 01239**
Phone (626) 395-2129	Phone (617) 253-4824
Fax (626) 304-9834	Fax (617) 253-7014
E-mail: info@ligo.caltech.edu	E-mail: info@ligo.mit.edu

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

Creating an e2e Primitive

Jeff Jauregui

July 31, 2003

Abstract

This document walks you through the process of creating your own primitive module for use in alfi and e2e. Sample code is provided so that a module can be generated with very few modifications.

1 Introduction

Before we begin writing code, let's introduce and/or review a few terms. "e2e" means the end-to-end time domain simulation program. Using what are known as "box" files, the simulation is told what to do. A box file is merely a text file (read by e2e at run time) that tells the simulation how to pass data among the various modules, or primitives. A primitve can be thought of as a fundamental object; for example, the "sine" module has inputs of frequency, amplitude, phase, and the current time, and outputs the resulting waveform for that particular time. Thus, a box file is nothing more than a series of these primitives linked together. (Note that box files are rarely created by hand: a program known as "alfi" (AdLib Friendly Interface) allows us to graphically generate box files by positioning modules on a grid and connecting them with virtual wires.) This document walks you through the steps needed to create a new primitive.

2 Overview

1. For a primitive named "object," create the object.cc and object.h files in the AdlibMM folder.

2. Create the text file object.prm in the same directory.

3. To register the new module with the e2e code, edit the file "modeler_base_register.cc."

4. To include the module code in the next build, edit "Makefile.am."

5. In the sample code, don't forget to replace every occurence of "example" with your object's name.

3 The Header File

Generally, a module inherits the public properties of the "primitive" class. The only member function you need to modify is "action()": it is in charge of performing the module's various tasks at each time step of the simulation. Whatever protected member variables need to be created depends solely on the inputs and outputs of the primitive.

3.1 Creating the Inputs

Variables can be input to the module in three ways: they can be read in from another module, from a text file (called a "par" file), and from a setting in a box file. We handle each case separately.

Suppose the module requires two real number inputs, in0 and in1. We then need to make four declarations in the protected region of the class. First, we need pointers to the actual input data:

adlib_real *in0, *in1;

(Note that adlib_real is the data type e2e uses for real numbers.) We also need variables to hold the default values for in0 and in1 in the event that no values are supplied:

adlib_real default_in0, default_in1;

To create a single parameter named param0, we declare it to be, for example, adlib_real in the class header. A setting is declared in the same way.

3.2 Creating the Outputs

Creating the outputs is even easier. Suppose we have two real outputs, out0 and out1. Then we need only make the declaration:

adlib_real out0, out1;

You'll be finished with the header file as soon as the $\boxed{\text{OBJECT\_INIT}}$ macro is defined. (See the example code.) Don't forget to pass your object's constructor as the second argument of $\boxed{\text{MI\_STATIC\_INIT}}$.

4 The Implementation File

In this section we discuss how to implement your specific module.

4.1 I/O Setup

Your module's constuctor automatically calls the constructor of its parent class, primitive. The five arguments this takes are, in order: name_arg, parent_arg, your module's name in quotation marks, number of inputs, and number of

outputs. Don't worry about the first two; they are taken directly from your module's constuctor's arguments.

Immediately following this point you must set the default values for the inputs and outputs. For the inputs use the syntax: `default_in0(0.0), default_in1(0.0)`.

For the outputs, we set the default values as: `out0(0.0), out1(0.0)`.

Within the constructor, we need to explicitly tell the program what variables should be treated as inputs, outputs, parameters, and settings. For an input, use the sample code below:

```
setup_input (0, data_ref( &default_in0,
    data_ref::Type_Real), (const void**)(&in0));
set_input_name(0, "in0");
```

The first line tells e2e that there is a real input whose default value is stored in default_in0. The second line registers the input's name, "in0", so that alfi will associate the variable in0 with the actual input of the primitive in a box file. The first parameter, 0, common to both commands is simply an index that tells e2e which input we are referring to. For the next input, an index of 1 would need to be passed, etc.

To implement a parameter, we use the code:

```
add_auxiliary(data_ref(&param0,
    data_ref::Type_Real), "paramName");
```

To create an internal setting that can be modified within alfi, use the same procedure as for parameters, except add the line `real setting = 0.0` to the *.prm file, where 0.0 is the default value if none is given. Note that the string passed to add_auxiliary must agree with the variable name used in the *.prm file.

The syntax for creating an output is straightforward:

```
setup_output (0, data_ref( &out0, data_ref::Type_Real));
set_output_name (0, "out0");
```

4.2   The Action Function

In essence, the action function sets values for the outputs in terms of the inputs, parameters, settings, and possibly other variables. In fact, it could be as simple as:

```
out0 = *in0 + *in1;
out1 = *in0 − *in1;
```

For a module such as this, the outputs would always hold the sum and difference of the two inputs at any time step of the simulation.

5 Registering and Building the Primitive

In the AdlibMM directory, open the file "modeler_base_register.cc". First, include your object's header file at the bottom of the other preprocessing directives using the line $\boxed{\text{\#include "object.h"}}$. Then add the line $\boxed{\text{OBJECT\_INIT(the\_modeler)}}$ after the similar lines in the sample code. Save and close the file.

Now, your primitive is registered with e2e. However, we need to instruct the compiler to include the new code in future builds. This is done through the text file "Makefile.am" in the same directory. Add your implementation file "object.cc" to the section $\boxed{\text{MODULE\_DYNAMIC}}$. Be sure to keep the existing convention of separating lines with a backslash. In the same manner, add your header file "object.h" under the line $\boxed{\text{noinst\_HEADERS}}$. Finally, add the alfi file, "object.prm" (even if you haven't created it yet), under $\boxed{\text{moddesc\_HEADERS}}$.

6 The ALFI File

Below is the code for the *.prm file corresponding to the module described above:

```
% Your description: out0 = in0 + in1, out1 = in0 – in1
%*Port input in0
%*{
%* dataType = real
%*}
%*Port input in1
%*{
%* dataType = real
%*}
%*Port output out0
%*{
%* dataType = real
%*}
%*Port output out1
%*{
%* dataType = real
%*}
%*GUI_Settings
%*{
%* ScreenSize 150x150
%* Group 'Whatever'
%*}
```

Appendix: Source Code

Header File:

```
/*
 * This is an example to create a module based on primitive class.
 * This module has two real inputs, two real outputs and one parameter.
 */

#ifndef __EXAMPLE_H__

#define __EXAMPLE_H__

// this class is based on primitive class
#include "primitive.h"


/*******************************##*******************************\
*
* example class
*
\*******************************##*******************************/

class example : public primitive
{
  public:
  // define constructor just this way
  example
  (const string& name_arg = "", const module* parent_arg = NULL);
  ~example(void);

  // this is used to create an object in the event queue
  module* new_type(const string& name_arg, const module* parent_arg) const;

  // Sets the simulation time step for this module.
  bool set_time_step(adlib_real time_step_arg);

  // this is called when settings are changed
  void sub_sub_load( void );

  // This is called repeatedly at each time step. This needs to update the output
  void action( void );


protected:
  // inputs
  adlib_real *in0, *in1;
  // storage of default inputs, if nothing is connected to input ports,
  // input ptrs point to these
  adlib_real default_in0, default_in1;

  // outputs
  adlib_real out0, out1;
```

```
  // settings
  adlib_real param;

  // private variable
  adlib_real val;

};  //end class

// this is needed for each header file
// MI_STATIC_INIT needs to be included for each function defined in this file
#if defined(engine_static)
#define EXAMPLE_INIT(a) \
  MI_STATIC_INIT(a,example());
#else
#define EXAMPLE_INIT(a)
#endif

#endif  // __EXAMPLE_H__
//end of header file
```

Implementation File:

```
// implementation of example class

#include "example.h"

/* ++++ from here to a line with ---- are needed for each file. */
// MI_KEY is a tag for this file, and can be anything unique
#define        MI_KEY example

#include "mi.h"

MI_FUNC()
{
  MI_DECL(1);

  MI_MODULE(example);  // add for each function defined in this file
  MI_EXIT();
}
/* ----- */

example::example
(const string& name_arg, const module* parent_arg)
  : primitive(name_arg, parent_arg,
             "example" /* name of primitive */,
             2 /* number of inputs */,
             2 /* number of outputs */ ),
    default_in0(0.0), default_in1(0.0),
    out0(0.0), out1(0.0),
    param(0.0), val(0.0)
```

```
{
  // define inputs, one for each input
  setup_input
    ( 0 /* serial index, starting with 0 */,
      data_ref( &default_in0 /* default storage if none connected */,
                data_ref::Type_Real /* data type defined in data_ref.h */),
      (const void**)(&in0) /* pointer to the input */ );
  set_input_name
    ( 0 /* serial index */,
      "in0" /* name of this input */);
  setup_input
    ( 1 /* serial index, starting with 0 */,
      data_ref( &default_in1 /* default storage if none connected */,
                data_ref::Type_Real /* data type defined in data_ref.h */),
      (const void**)(&in1) /* pointer to the input */ );
  set_input_name
    ( 1 /* serial index */,
      "in1" /* name of this input */);

  /* define outputs, one for each output */
  setup_output
    ( 0 /* serial output index, starting with 0 */,
      data_ref( &out0 /* output variable */,
                data_ref::Type_Real /* data type */));
  set_output_name
    ( 0 /* serial index */,
      "out0" /* name of this input */);

  setup_output
    ( 1 /* serial output index, starting with 0 */,
      data_ref( &out1 /* output variable */,
                data_ref::Type_Real /* data type */));
  set_output_name
    ( 1 /* serial index */,
      "out1" /* name of this input */);

  // defining user settings, one for each setting
  add_auxiliary
    ( data_ref(&param /* storage of the setting value */,
               data_ref::Type_Integer /* data type defined in data_ref.h */),
      "paramName" /* name of setting */);
}

example::~example( void )
{}

// implement this just as it is
module* example::new_type
(const string& name_arg, const module* parent_arg) const
{ return new example(name_arg, parent_arg); }

bool example::set_time_step(adlib_real time_step_arg)
{
```

```cpp
  if ( time_step_arg < 1e-10 )
    {
      mess(WARNING) << "time step too short : time step = "
                    << time_step_arg << endl;
      return false;
    }
  else
    {
      return true;
    }
}

// this is called when a setting is changed
void example::sub_sub_load( void )
{
  if ( param < 0 )
    {
      mess(ERROR) << "param should be positive or 0" << endl;
      exit(1);
    }
  else if ( param == 0)
    {
      mess(WARNING) << "param = 0 is fragile" << endl;
      val = 1;
    }
  else
    {
      val = 1/param;
    }
}

// this is the core part
void example::action( void )
{
  // calculate outputs using inputs and local variables
    out0 = *in0 + *in1;
    out1 = *in0 - *in1;
}
```