# Creating GLM Files

From GridLAB-D Wiki (UVic shadow)

**TODO:** Add Hassayampa (Version 3.0) directives

This page focuses on the `.glm` file format. For information on the format of `.xml` files, please consult the XML_Data_Files page.

A `.glm` file is used to synthesize a population of objects. This is in contrast to an `.xml` file, which is used to represent a specific population that has been synthesized. Consider the difference between the following `.glm` file that define a house object as

```
object house {
  floorarea random.normal(2000,500) sf;
}
```

meaning that a single house is to be defined, and the `floorarea` is to be normally distributed with a mean of `2000 sf`, and a standard deviation of `500 sf`.

## Contents



In contrast, the relevant section of the output `.xml` might look like

```
<house>
  <floorarea>
    <unit>sf</unit>
    1853.56
  </floorarea>
</house>
```

meaning that house id `0` has a `floorarea` of `1853.56 sf`. Notice that the `.xml` no longer contains any information about the distribution. However, if we were to generate a large population of houses, either by running GridLAB-D many times with this `.glm` house definition, or by defining a large number of houses in a `.glm` single model, we could observe that the normal distribution specified in `.glm` file is implicit in the `.xml` file that is output.

# Comments

Comments begin with a `//` sign. All text between the `//` and the end of the line is ignored by the parser.

It is important to note that the parser in GridLAB-D is quite primitive and may in certain circumstance be confused by `//` appearing in other contexts, such as a URL not enclosed in a quotes.

# Macros

The `.glm` loader allows the use of macros to control the behavior of the parser and to a limited extent also the behavior of GridLAB-D. Macros are lines that begin with a '#' sign. The following macros are available

- `#define` *`name=value`* is used to define a global variable. This allows the creation of a new global variable, in contrast to the `#set` macro which requires the global variable already exist.
- `#set` *`name=value`* is used to set a global variable. For a list of defined global variables, see the [Doxygen Documentation (http://www.gridlabd.org/documents/doxygen/latest_dev/globals_8h.html)].
- `#undef` is used to remove the definition of a global variable.
- `#ifdef|ifndef <expression>`
  ...
  [`#else`
  ... ]
  `#endif` is used to conditionally process a block of text in the `.glm`. For `#ifdef` and `#ifndef` the expression is simply the name of a global variable. When `#if` is used, the expression is a conditional test in the form of *`name op value`* where the operator *op* is one of `<`, `>`, `<=`, `>=`, `==`, or `!=`.
- `#include` *`file`* is used to include text from *`file`* in the loaded text.
- `#print` *`name`* is used to display the value of the global variable `<name>` at the moment it is encountered by the loader.
- `#ifexist` *`file`* is used to determine whether a file exists
- `#include` *`file`* is used to include another file during the parser load
- `#setenv` *`name=string`* is used to set an environment variable
- `#binpath` *`path`* sets the path for binary searches (compiler PATH environment)
- `#libpath` *`path`* sets the path for library seaches (compiler GLPATH environment)
- `#incpath` *`path`* sets the path for include file searches (compiler INCLUDE environment)
- `#error` *`message`* triggers a parser error condition
- `#warning` *`message`* triggers a parser warning condition
- `#option` *`command-option`* applied the command line option
- `#system` *`command`* makes an operating system call and waits for completion before continuing processing the GLM file
- `#start` *`command`* makes an operating system call and continues processing the GLM file

# Modules

Static modules must be loaded before the classes they define can be used or modified. To load a static module, use `module` *`name`*`;`. The module loader will search the **`GLPATH`** environment variable to locate a file named *`name`*`.dll` (on linux the search will be for `lib`*`name`*`.so`. If a particular version is desired, the version must be appended to the name using the format `module` *`name-major`*`;` or `module` *`name-major_minor`*`;` depending on whether you wish to specify on the major version, or both the major and minor version.

Module blocks may include additional information, such as assignments of the values for module globals and specification of the version number. To set a module global variable, simply include the name and value in the module block, such as

```
module MyModule {
  MyStringGlobal "value";
```

```
  MyEnumGlobal A;
  MyDoubleGlobal 1.2 ft/s;
}
```

To enforce verification of the module's version information, simply include the desired version in the module block:

```
module MyModule {
  major 2;
  minor 1;
}
```

If the version loaded does not match the version specified, an error will be displayed and the loader will stop.

# Classes

Class blocks are used to create, modify, or verify class definitions. If a class is already defined in a static module, then a class block either modifies or verifies the definition provided by the module. Consider the following example

```
module MyModule;

class MyClass {
  char32 svalue;
  enum {A=0,B=1,C=2} evalue;
  double dvalue[W];
}
```

If the properties `svalue`, `evalue`, and `dvalue` are already defined as specified, the class block will load successfully. However, if there are any differences between the class block and the module's definition of the class, then the loader will attempt to address the discrepancy as follows:

1. If the class defines a property differently than the module, then the loader will fail.
2. If the class defines a property that the module does not define, then the loader will extend the module's definition of the class to include this new property.
3. If the class is not defined by any loaded module, then the class is defined as a new class, and the properties are added to that new class. In this case, you may also include C++ code for the behaviors that static modules normally provide. See below for more information on runtime classes.

# Runtime Classes

If a class block is defined, and that class is not already implemented by an existing module, then you may define class behavior for each of the supported behaviors in GridLAB-D. Such classes are called *runtime classes* and are only be supported if you have installed MinGW on your system. Use of the compiler is automatic and does not need to be explicitly configured. However, if you have installed GridLAB-D in an unusually location, you may have to set the global variable `INCLUDE` to indicate where the `rt/gridlabd.h` file is located. You may also want to add the location of any files that the `#include` macros are expected to find.

The supported behaviors for runtime classes include:

- `create (object <parent>) { ... return [SUCCESS|FAILED];};` is used to define the object creation behavior other than the default behavior (which is to set the entire object's memory buffer to 0). Usually *`parent`* is not defined at this point. Object creation functions are called when the object's memory is allocated, while initialization isn't performed until after all objects have been created and the parent/child hierarchy established.
- `init (object <parent>) { ... return [SUCCESS|FAILED];};` is used to define object initialization behavior, which is delayed until all objects have been created and their properties set as defined in the model file. Usually initialization is a good time to adjust dependent properties and/or check for inconsistencies in the values provided.
- `[presync|sync|postsync] (TIMESTAMP t0, TIMESTAMP t1) { TIMESTAMP t2 = TS_NEVER; ... return t2;};` is used to implement **presync**, **sync**, and **postsync** behavior. Presyncs are called on the first top-down pass, syncs are called on the bottom-up pass, and postsyncs are called on the last top-down pass. The top-down/bottom-up order in which objects are evaluated is based on their ranks. Rank is determined primarily by the parent-child relationship, however calls to **`gl_set_rank()`** can be used to promote the rank of an object arbitrarily with respect to another object. (Objects cannot be demoted.)
- `plc (TIMESTAMP t0, TIMESTAMP t1) {TIMESTAMP t2 = TS_NEVER; ... return t2;};` is used to define the default programmable logic controller (PLC) behavior. This behavior is overridden if a `plc` refers to this object as its parent.

Over time other behaviors, such as `check`, `import`, `export`, `kmldump`, etc. will be added, as needed.

For more information on the compilers for runtime classes, see the page on Runtime compiler support.

# Exporting functions from runtime classes

You can publish a function from a runtime class by using the following syntax.

```
class example1 {
  export myfunction(void *arg1, ...)
  {
     // your code goes here
     return result; // int64 value is returned
  }
}
```

and access that function from another class using the syntax

```
class example2 {
   function example1::myfunction;
   intrinsic sync(TIMESTAMP t0, TIMESTAMP t1)
   {
      // your code goes here
      int64 result = myfunction(arg1, arg2);
      return t2;
   }
```

# Debugging Runtime Classes

To debug runtime class behavior, you must install **gdb** on your system. The debugger does not appear to be fully functional at this time. As more is learned about its limitations and workarounds, tip and tricks will be posted at Runtime debugging.

Windows users can debug with MS Visual Studio 2005™. To enable this debugger the following environment variables should be set

```
// set this to your MSVC installation folder
#setenv MSVC=C:/Program Files/Microsoft Visual Studio 8

// enable use of MSVC instead of GNUtools
#define use_msvc=1

// this disables automatic rebuild suppression of runtime classes
// compilation based on modification time
#set force_compile=1

// customize to your local setup
#setenv path=${MSVC}/Common7/IDE;${MSVC}/VC/bin
#setenv include=${MSVC}/VC/include
#setenv lib=${SystemRoot}/system32;${MSVC}/VC/lib;${MSVC}/VC/PlatformSDK/Lib
```

You can get all these settings by simply including the debugger configuration file:

```
#include "debugger.conf"
```

You should set the debug program to the debug version of `gridlabd.exe` which is usually placed in the `$GRIDLABD/VS2005/win32/debug` folder. Once you have done this, you can set breakpoints in the C++ code block of your `.glm` files and the debugger will stop and offer most of the usual debugging features needed to debug your runtime class.

# Objects

Object blocks are used to define one or more instances of a class. The simplest form is a singleton object definition, such as

```
module residential;
object house {
  floorarea 2500 sf;
}
```

which will define an anonymous house of 2500 square feet.

If however, you wish to define 10 such houses, then you can specify a count, such as

```
module residential;
object house:..10 {
  floorarea 2500 sf;
}
```

If a functional value is used, then the functional value is evaluated separately for each instance created, so that

```
module residential;
object house {
  floorarea random.normal(2500,100) sf;
}
```

defines 10 separate house, each with a different floor area, but with the floor area normally distributed about a

mean of 2500 sf with a standard deviation of 100 sf.

If a mathematical expression is written within a set of parentheses, it will be processed at load time. Addition, subtraction, multiplication, division, exponential, and modulo operators are supported, as well as parenthetical nesting and unary math.h functions. The code

```
module residential;
object house {
  floorarea random.normal(2500,100) sf;
  ceilingheight random.normal(8,1) f;
  airvolume (this.floorarea * this.ceilingheight);
}
```

will generate a random value for the floor area and the ceiling height of a house and calculate the contained air volume.

# Expanded values

An expression written in back quotes will be parsed in such a way that expressions in curly-braces are expanded in the context of the object being loaded. The following values are expanded

- `{file}` embeds the current file (full path,name,extension)
- `{filename}` embeds the name of the file (no path, no extension)
- `{fileext}` embeds the extension of the file (no path, no name)
- `{filepath}` embeds the path of the file (no name, no extension)
- `{line}` embeds the current line number
- `{namespace}` embeds the name of the current namespace
- `{class}` embeds the classname of the current object
- `{id}` embeds the id of the current object
- `{var}` embeds the current value of the current object's variable *var*

For example,

```
namespace space1 {
        object mytest {
                name `{namespace}::{class}:{id}`;
        };
}
```

will result in the object having a name like `space1::mytest:0`.

# Object properties

The valid properties for are determined by the class definition. This includes the units specification. When units are specified in the class, then the value is always stored in memory in those units. If the object definition provide the value with different (but compatible) units, then the value is automatically converted to the units specified by the class so that the behavior needn't consider units when doing calculation using the value. Consider the following example

```
class house {
  double floorarea[sf];
```

```
  init (object parent) {
    printf("floor area is %f sf\n", floorarea);
  };
}
object house {
  floorarea 100 m^2;
}
```

When `init` is executed, the floor area will be displayed in **sf** and not **m^2** because the loader automatically converts the value to the units specified by the class definition.

# Parameter expansions

String, double, set and enumeration variables can be manipulated during expansion using parameter expansions. For example

```
#define AREA=1000
class house {
  double floorarea[sf];
}
object house:..5 {
  floorarea ${AREA+=100};
}
```

will create 5 houses with values of floorarea ranging from 1000 to 1400 by increments of 100.

Retrieved from "http://gridlabd.me.uvic.ca/wiki/index.php?title=Creating_GLM_Files&oldid=7647"

---

- This page was last modified on 28 June 2015, at 08:42.