U.S. DEPARTMENT OF ENERGY

PNNL-22611

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

The Parameter Manager Library

DES-0016

Revision 3

Charlie Hubbard

August 2012

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by **Battelle** *Since 1965*

The Parameter Manager Library

Charlie Hubbard

DES-0016
Revision 3
August 2012

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under
Contract DE-AC05-76RL01830

Printed in the United States of America

Contents

1 Introduction

This document describes the *Parameter Manager Library*, a collection of code designed to manipulate what amounts to fancy tag/value pairs. The library was developed specifically to provide *run-time parameter* support to the system's *control server*[1], but has since been adopted by other servers and utilities as well. The library offers the following features:

- Tag/value pairs (actually *parameters* since each parameter contains more information than just a value) are maintained in a collection called a *parameter set*
- The data associated with a specific tag is actually much more than just a value. Besides the value, there is a default value for the parameter, the engineering units the value is reported in, a time stamp indicating when the parameter's value was last modified, and a text description of the parameter.
- Internally, parameter names, their values and most of their associated data are maintained as strings. The one exception to this is the last modified time stamp, which is stored and returned as a Unix `time_t` type.
- Individual parameters are referenced by name (the tag), and their corresponding values can be rendered as a variety of different data types including bools, ints, doubles and strings.
- Parameter sets can only be initialized from text-based configuration files. Changes made to a set can be written back to the same configuration file as required by the application.
- Tag names are not case-sensitive, but they are stored internally in all uppercase, and that is how they are returned to the user or written back to the configuration file. Values (and their associated attributes like engineering units) *are* case sensitive, and are maintained exactly as they were provided.
- Internally, parameters are maintained in an *Standard Template Library* (STL) map, making access to them very fast.

The Parameter Manager Library is defined and implemented by the two files *paramsManager.h* and *paramsManager.cpp*. The primary documentation for the library is its source code and its associated Doxygen-generated HTML files. The Doxygen documentation should be consulted for a detailed description of the library's *Application Programming Interface* (API). The document you are reading now is supplemental, and is intended to provide deeper background for the library's implementation. In cases where the Doxygen documentation disagrees with this document, the Doxygen documentation should be considered correct.

[1]See design document ***DES-0015, The Control Server*** for details on the control server application.

2 Library Classes

The library implements parameter-set support with three classes called `ParameterFields`, `Parameter` and `ParameterSet` respectively.

2.1 The `ParameterFields` Class

The `ParameterFields` class defines the data fields associated with a single parameter as well as a constructor and certain utility methods. The class definition appears below.

```
struct ParameterFields {
   ParameterFields();

   string name;
   string value;
   string defval;
   string units;
   time_t lastModified;
   string description;

   void   Dump() const;

   string Serialize() const;
   void   Deserialize(const string s);
};
```

Parameter fields are sequestered away in their own structure like this, rather than being placed directly in the `Parameter` class (see the following section), mostly as a matter of style, with the following justifications.

- Of a parameter's various fields, only the `value` field can be manipulated directly by the user. Other fields can only be populated at the time the `ParameterSet` configuration file is processed. That is to say, other than the value field, a parameter's fields are in effect *read-only.* If the fields were direct attributes of the `Parameter` class, they could not be read-only if they were made *public* so they'd have to be *private* or at least *protected.* But if the fields were not public, then a series of accessor methods (one for each field) would be needed to make them retrievable by the user, but not settable. Instead, in the actual implementation, each `Parameter` contains one private instance of `ParameterFields` and a single accessor method that returns the entire structure.
- The field data for a parameter in a parameter set are associated with the parameter only when the parameter set is first initialized (that is, when the parameter set configuration file is processed). During this process, the field data are passed into the `Parameter` class constructor. In the author's opinion, it is cleaner to pass the constructor a single structure containing the fields instead of a list of individual field values (this becomes more and more of an issue as the number of fields increases).
- Over time, the number of fields associated with a single parameter has grown from two (`name` and `value`) to the list that exists today. There is every reason to believe that

some future enhancement of the library will result in even more fields. If this happens, the above two points remain the same—each `Parameter` class will still only require one accessor method to allow users to get at the field data, and the `Parameter` constructor parameter list will not have to expand to accommodate additional fields.

Some further explanation is probably warranted for the `Serialize()` and `Deserialize()` methods. These two methods exist to support passing `ParameterField` classes between clients and servers using our standard client/server message-passing mechanism[2]. As you'll recall, under our architecture, all messages exchanged between clients and servers are text strings encoded using our *Data Serialization Protocol*[3] (DSP). The `Serialize()` method returns a DSP compatible string containing the field data. Similarly, the `Deserialize()` method, takes a DSP string as returned by `Serialize()` and uses the data encoded within to populate the structure's field members.

2.2 The `Parameter` Class

The `Parameter` class defines a single parameter. Its definition appears below.

```
class Parameter {
public:
   Parameter();
   Parameter(const ParameterFields &fields);
   string          GetName() const;
   bool            AsBool() const;
   int             AsInt() const;
   double          AsDouble() const;
   char            AsChar() const;
   string          AsString() const;
   ParameterFields GetFields() const;
   void            Dump() const;
   void            RevertValueToDefault();
   void            SetValue(const string v);

   string Serialize() const;
   void   Deserialize(const string s);

private:
   ParameterFields fields;
};
```

As mentioned previously, a parameter's value (its `value` field) is stored internally as an STL `std::string`, but of course it may be interpreted by the user in any way that makes sense to the application. As a convenience, a series of methods exist to render the internal string in a variety of common data types including `std::string`, `bool`, `int` and `double`. These are the `As...()` methods shown above.

`GetName()` is a convenience method, the functionality of which is somewhat redundant since the user can also get the parameter's name by retrieving all of the parameter's field data

[2]See design document ***DES-0005*, *The Client/Server Architecture*** for a complete description of client/server message-passing as implemented in our architecture.

[3]See design document ***DES-0002*, *The Data Serialization Protocol*** for a complete description.

using the `GetFields()` method. However, experience has shown that access to a parameter's name by itself is required commonly enough to warrant the redundancy.

The `GetFields()` method is the primary field data accessor method described in the previous section. It simply returns the private `ParameterFields` element `fields`.

The `Serialize()` and `Deserialize()` methods perform the same function as the methods by the same name described in the previous section.

2.3 The `ParameterSet` Class

Individual parameters are stored together in a *parameter set* implemented by the `ParameterSet` class. A copy of its definition appears below.

```
class ParameterSet {
public:
   ParameterSet();
   void       Clear();
   void       Dump();
   bool       Exists(const string n);
   int        GetNumParams();
   void       RevertValuesToDefaults();
   Parameter& operator[](const string n);

   void       Reset();
   Parameter  GetNext();

   string    Serialize() const;
   void      Deserialize(const string s);

   string  InitializeFromConfigFile(const string fname);
   string  RewriteConfigFile();

private:
   typedef map<string, Parameter> ParameterMap;

   string                 configFileName;
   ParameterMap           parameterMap;
   ParameterMap::iterator  pmItr;
   Parameter              emptyParam;
};
```

The class provides methods for

- initializing the parameter set with parameter data read from a configuration file (this is done using the `InitializeFromConfigFile()` method)
- removing all parameters from a parameter set (`Clear()`)
- testing for the existence of a specific parameter by name (`Exists()`)
- modifying the values of individual parameters in the set (this is actually done indirectly through the `[]` operator and the `Parameter` class' `SetValue()` method)
- accessing individual parameter values by parameter name (the `[]` operator)

- iterating through all parameters in the set (`Reset()` and `GetNext`)
- writing parameter values back to the configuration file that was used to originally populate the set (`RewriteConfigFile()`)
- and, of course, serializing and deserializing the contents of the parameter set into and out of DSP format as described previously (`Serialize()` and `Deserialize()`)

The `[]` operator is particularly important, because it is the primary means by which a user will access individual parameters in a parameter set. It takes the name of a parameter as its input, and it returns a reference to the corresponding `Parameter` class. It is important to note however, that the `[]` operator does not error or throw an exception if the requested parameter is not part of the set. Instead it silently returns a reference to an internal empty parameter (its string value is an empty string, its boolean value is *false*, and its numerical value is zero). It is also through the `[]` operator that users modify a parameter's value. This is done indirectly via the `Parameter` class' `SetValue()` method. This is demonstrated by example in the following code snippet.

```
set["MY_PARAM_NAME"].SetValue("1.234");
```

The `Reset()` and `GetNext()` methods together provide the mechanism for traversing the parameters in the set. Calling `Reset()` resets the internal traversal pointer back to the beginning of the set[4]. Afterward, each successive call to `GetNext()` returns a copy of the next parameter in the set until all parameters have been returned. Parameters are returned in alphabetical order by name. The user should use the `GetNumParams()` method to determine how many times `GetNext()` needs to be called to traverse the entire set. If `GetNext()` is called more than this many times, empty parameters will be returned.

3 The `ParameterSet` Configuration File

The configuration file read and written by the `ParameterSet` class, has a very simple format defined by two simple rules:

- Blank lines and lines beginning with a '#' character are ignored. All other lines are parameter definition lines.
- Parameter definition lines have five or six fields separated by one or more whitespace characters. The fields are: 1) parameter name, 2) parameter value, 3) parameter default value, 4) engineering units the parameter is reported in, 5) the value's last-modified time stamp, and 6) an optional free-form text description of the parameter.

[4]Currently sets are implemented as STL maps, so this amounts to setting a private map iterator to `map.begin()`, but the map is hidden from the user, and could easily be replaced by a different data structure in the future if warranted.

New configuration files are expected to be created by using any suitable text editor like *gedit*, *emacs* or *vim*.

When manually creating a new configuration file, the last-modified timestamp field is normally set to zero. This field actually contains a Unix `time_t` type integer[5]. These values are cumbersome to generate manually and are not human-readable. By convention, we use a value of zero to mean "has never been modified." However, users are certainly welcome to enter legitimate `time_t` values if they prefer.

Configuration files, by convention, typically begin with a comment header block that describes the file, but comment lines and blank lines can be used internally in any way the author chooses to help organize the file's contents. If the current contents of a parameter set are later pushed back to the set's configuration file (via the `RewriteConfigFile()` method), the content and order of the comments and blank lines will be preserved.

A sample configuration file is shown below

```
##########################################################################
#
# This configuration file is compatible with the Parameter Manager Library
# (see paramsManager.h/.cpp and design document DES-0016 for details).
# The following parameters constitute the run-time parameters used by the
# project X control server.
#
##########################################################################

NUM_SAMPLES_PER_POINT     5    25   none  0  Record this many samples to the output file
CARRIER_CODE              0     0   none  0  Carrier gas code
GAS_CODE                 23    23   none  0  Sample gas code
POST_LOAD_DELAY           5    20      s  0  Delay this much time to reach equilibrium
POST_PUMP_DOWN_DELAY      5   300      s  0  After system pump-down, delay this many additional seconds
PUMP_BETWEEN_SAMPLES      0     0   bool  0  If 1, then pump down between samples
PUMP_DOWN_PRESSURE      1.5   1.5   Torr  0  Sensors pumped down to this pressure between samples
SET_SPANS                 0     1   bool  0  If 1, then the sensors will have their zeros and spans set
SPAN_PRESSURE           760   760   Torr  0  This is the pressure that will be used to set the gauge spans

##########################################################################
```

4 A Parameter Set Example

Here we show a very simple example of the Parameter Manager Library in action. In this example, we'll initialize our parameter set from a configuration file, and we'll assume the configuration file is the same as the one shown above. In this example we first print out the values for a few specific fields to demonstrate how parameters are accessed with the `[]` operator and how the type converters work. Then we use the set traversal mechanism to print out the values of every parameter in the set.

[5]These record the number of seconds that have elapsed since midnight of January 1st, 1970 GMT.

```
#include <iostream>
using namespace std;
#include "paramsManager.h"

int main()
{
   ParameterFields  pfields;
   ParameterSet     ps;

   ps.InitializeFromConfigFile("ourfile.cfg");

   cout << "Current post-load delay = " \
        << ps["POST_LOAD_DELAY"].AsInt() \
        << " seconds." \
        << endl;

   cout << "Current pump-down pressure '" \
        << ps["PUMP_DOWN_PRESSURE"].AsDouble() \
        << endl;

   ps.Reset();
   for (int i=0; i< ps.GetNumParams(); i++) {
      pfields = ps.GetNext().GetFields();
      cout << "Name=" << pfields.name \
           << ", Value=" << pfields.value \
           << ", Default=" << pfields.defval \
           << ", Units=" << pfields.units \
           << ", LastModified=" << pfields.lastModified \
           << ", Description='" << pfields.description<< "'" \
           << endl;
   }

   return(0);
}
```