U.S. DEPARTMENT OF ENERGY

PNNL-22596

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

The Output Channel Cache Library

DES-0017

Revision 1

Charlie Hubbard

June 2011

Pacific Northwest
NATIONAL LABORATORY

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

The Output Channel Cache Library

Charlie Hubbard

DES-0017
Revision 1
June 2011

DISCLAIMER

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

Available to DOE and DOE contractors from the Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831-0062
ph: (865) 576-8401
fax: (865) 576-5728
email: reports@adonis.osti.gov

Available to the public from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161
ph: (800) 553-6847
fax: (703) 605-6900
email: orders@ntis.fedworld.gov
online ordering: http://www.ntis.gov/ordering.htm

Contents

1 Introduction **1**

2 Library Classes **2**
2.1 The `Cache` Class 2
2.2 The `CacheManager` Class 3

3 The Library Configuration File **5**

4 Some Coding Examples **6**

1 Introduction

This document describes the Output Channel Cache Library. The library is defined and implemented in *outputChannelCache.h* and *outputChannelCache.cpp* respectively. The primary documentation for this library is its corresponding Doxygen-generated HTML output. The Doxygen documentation should be consulted for a detailed description of the library's API. The document you are reading now is supplemental, and is intended to provide deeper background for the library's purpose and implementation. In cases where the Doxygen documentation disagrees with this document, the Doxygen documentation should be considered correct.

The output channel cache library was written as a support library for the system's control server[1]. It serves as an intermediate layer between the control server application and the hardware outputs the control server manipulates.

The control server uses an internal state machine based on our State Machine Library[2]. One feature of that library is the option to allow a user to manually force jumps from any state in the state machine to any other state in the state machine. However, doing so in real-world applications often requires special planning. In the case of the control server's state machine, the ability to arbitrarily jump into any state means that each state must be able to fully configure the system's hardware control outputs as appropriate for that state. States cannot rely on earlier states to have already done this, because there is no guarantee that earlier states in the normal transition chain have actually been visited. But the system has many dozens of hardware control outputs (valve controls, heater setpoints, gas flow rate setpoints, etc.). Fully specifying the state of *every* control output in the entry function of every state machine state would be both cumbersome and error-prone. It is these issues the output channel cache library was designed to address.

The library uses a configuration file to store the names of every hardware output channel it is allowed to manipulate, along with default values for those channels. The purpose for maintaining default values is this: when writing the entry function for a given state machine state, the author need only specify the output state of those hardware channels that will differ from their default values. Typically this is a very small subset of the total number of outputs. All other output channels will automatically be set to their default values. This feature enormously simplifies the task of the state machine author.

The library also maintains an internal cache, which stores the current state of each output. When the control code requests that an output channel be configured to a certain value, that value is first compared with the value in the cache. If they match (meaning the channel is already configured to that value), then no client request is sent to the server that maintains that output. This can significantly reduce the amount of client/server message traffic between the application using the output channel cache library and the servers the library communicates with.

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

[2] See design document ***DES-0014, The State Machine Library*** for a better understanding of features and limitations of the state machine used by the control server.

Of course external clients can instruct a hardware server to change the value on a hardware output unbeknownst to the output channel cache library. For that reason, the library also provides a synchronizing ability, so it can be instructed to update its cached values with the actual values that currently appear on the control outputs. When this ability is used, depends on the application. The project's control server can resynchronize at any time via client request, but it also resynchronizes with every state change.

Currently three types of hardware output channels are supported — digital outputs (valves, relays, etc.), analog outputs (flow and pressure setpoints, etc.), and PID loops (special-purpose outputs associated with the Watlow CLS16 multi-channel PID controller typically used for heater temperature control and liquid nitrogen level control). Other types of hardware outputs (if there are any) must always be explicitly specified for every state, and any mechanism to mitigate client/server traffic for these outputs must be developed externally (or, more often, just ignored). Thankfully, output channels of a type not described above are rare.

2 Library Classes

The output channel cache library defines two classes called `Cache` and `CacheManager` respectively. These are described in detail in the following sections.

2.1 The `Cache` Class

The `Cache` class is a helper class to the `CacheManager` class described in the next section. No instance of the `Cache` class is used directly by users of the library.

The `Cache` class is used, perhaps obviously, to maintain the current values for all the hardware output channels currently being tracked by the library. It is this cache that the library compares against when deciding whether or not a client request needs to be sent to the corresponding hardware server to set an output channel to a particular value. However, the `Cache` class has other uses as well. One instance of the class is used to maintain the default values associated with each output channel, and another instance is used to hold a group of new channel values that are to be pushed out to the hardware all at one time (more on this in the following section).

The `Cache` class definition appears below. For a detailed description of each method and attribute, see the class' associated Doxygen documentation.

```
class Cache {
public:
   Cache();
   Cache(Cache const &c);
   Cache & operator = (Cache const &c);
   ~Cache();
   void Clear();
```

```
   void Dump();
   DigitalMap digitalMap;
   AnalogMap  analogMap;
   LoopMap    loopMap;
};
```

There are three *Standard Template Library* (STL) maps (keyed on output channel name) that maintain the value for every output channel tracked by the library. The values stored in these maps can be the current value of the output channel, the new value the user wants the channel to be set to, or the default value of the channel, depending on which instance of the class is being addressed.

The class really only has two methods — `Clear()` and `Dump()`. `Clear()`, as its name implies, removes all channels from the internal maps. The `Dump()` method dumps the contents of each of the three maps to the console. `Dump()` exists primarily for testing and debugging purposes.

You'll notice that there are no methods for adding new values to the maps. Values are added to the class' maps directly by methods in the `CacheManager` class. Users of the library do not manipulate the maps directly, however. Instead they call the appropriate `CacheManager` methods.

2.2 The `CacheManager` Class

The `CacheManager` class provides the API to the library. It provides methods for initializing the channel cache either directly from code or from a configuration file (or a combination of the two), and it provides the interfaces a user needs to set a hardware channel to a specific value or to synchronize the internal cache with the current state of the actual hardware. The class definition appears below.

```
class CacheManager {
public:
   CacheManager(AnalogClient *ac,
                DigitalClient *dc,
                LoggerClient *lc,
                PIDClient *pc,
                const string configFile = "");

   void AddDefaultAnalog(const string name, const double value);
   void AddDefaultDigital(const string name, const bool state);
   void AddDefaultLoop(const string name, const double setpoint, const bool enabled);
   void ReadConfigFile(const string configFile);

   void AddDigital(const string name, const bool state);
   void AddAnalog(const string name, const double flow);
   void AddLoop(const string name, const double setpoint, const bool enabled);

   void SetDigital(const string name, const bool setting);
   void SetAnalog(const string name, const double flow);
   void SetLoop(const string name, const double setpoint, const bool enabled);

   void ClearNewValues();
   void PushNewHardwareSettings();
```

```
   void SyncCache();

private:
   AnalogClient   *analog;
   DigitalClient  *digital;
   LoggerClient   *logger;
   PIDClient      *pid;
   string          configFileName;
   Cache           defaults;
   Cache           currentValues;
   Cache           newValues;
};
```

Because the library serves as an interfacing layer between the application (typically the control server) and the hardware servers, it must necessarily maintain client interfaces to those servers responsible for the hardware channels the library manages. It expects the application to supply those interfaces. Pointers to them are passed in to the class' constructor. The constructor also optionally takes the name of a configuration file the library will use to initialize its default cache.

The default cache is a private instance of the `Cache` class called `defaults`. Although it is typically populated from a configuration file by the `ReadConfigFile()` method, values can also be added to the default cache via the `AddDefault...()` methods. There is one of these methods for each type of output channel supported by the library.

When a user needs to set the value of an output channel, there are two mechanisms to pick from. The simpler one is to use the `Set...()` methods. These methods compare the new value for the channel with the value the channel already has (or at least supposedly has, according to the `currentValues` cache), and, if they are different, generates a client request to the appropriate hardware server to update the channel's output value.

This mechanism is simple, and because of its built-in ability to reduce client/server messaging traffic, it is many times also quite useful. However, with this mechanism alone, a state machine author would still be reduced to explicitly specifying the values for *every* output channel in the entry functions of *every* state. For that reason, the more complicated of the two channel update mechanisms is most often used instead.

The second mechanism involves specifying the intended values for a number of output channels as a set, and then pushing all the changes down to the hardware at one time. A private instance of the `Cache` class called `newValues` is used in this process, which works like so:

1. The user calls `ClearNewValues()`, which really does nothing but clear out the `newValues` cache.

2. The user makes as many calls to the `Add...()` set of methods as necessary to set up the hardware output channels he wants to configure. During this process, there is no need to specify an output channel unless its value is to be something other than its default value. All other channels will automatically be set to their default values in the next step. Experience has shown that, in the overwhelming majority of cases, the number of outputs that need to be different from their defaults is tiny compared to the

total number of outputs, so this mechanism can enormously reduce the coding effort required of the state machine's author.

3. Finally the user calls `PushNewHardwareSettings()`. This method iterates through each entry in the `defaults` cache, checking to see if the entry also exists in the `newValues` cache. If it does, then the value in the `newValues` cache is set; otherwise the value in the `defaults` cache is set. "Set" as used here actually means a call to the appropriate `Set...()` method as described above, meaning no actual client message is sent to a hardware server unless the real hardware value is different than the new value.

3 The Library Configuration File

As has been discussed previously, the library is assigned the hardware output channels it is responsible for either via configuration file, or directly in code (via the `AddDefault...()` methods). In this section, we look at the format of the library's configuration file.

The configuration file used by the library is a simple, text-based file with a syntax described by a few simple rules.

- Blank lines and lines beginning with a "#" character are ignored. All other lines are hardware output channel definition lines.
- By convention, the configuration file should begin with a comment block identifying the file as an Output Channel Cache library configuration file, along with any other bits of information the author may want to pass along. The beginning comment block is useful but optional.
- Three types of hardware definition lines exist, one for each type of output channel supported by the library. They are:

  1. Digital output channel definition lines, which have the following syntax:

     name `dout` *default*,

     where *name* is the channel name, `dout` is the literal string "dout" (without the quotes), and *default* is the default value for this channel. In this case, valid values are "on" and "off" (without the quotes).

  2. Analog output channel definition lines, which have the following syntax

     name `aout` *default*,

     where *name* is the channel name, `aout` is the literal string "aout" (without the quotes), and *default* is the default value for the channel. In this case, valid values are any floating point value supported by the channel.

  3. PID output channel definition lines, which have the syntax shown below. Notice that PID channels have two values that define their state—one to specify the

setpoint value for the control loop, and one to indicate whether or not the loop is enabled or disabled.

name `loop` *default-setpoint default-state*,

where *name* is the channel name, `loop` is the literal string "loop" (without the quotes), *default-setpoint* is the default setpoint value to use on this control loop (any floating point value within the allowed range for the loop) and *default-state* is the default enabled state for the control loop (either "on" or "off," without the quotes).

- By convention, lines of a given type are grouped together in sections, although this is not a requirement. Lines of different types can be freely interspersed.

An example configuration file is shown below.

```
################################################################################
#
# This is the control channel default values configuration file used by
# the control server.  It is compatible with the output channel cache
# library.  See outputChannelCache.h/.cpp or design document DES-0017 for
# more details.
#
#-------------------------------------------------------------------------------
#                          Digital Control Outputs
#-------------------------------------------------------------------------------
V100      dout  off
V101      dout  off
V200      dout  off
V201      dout  off
V300      dout  off
blower    dout  off
chiller   dout  off

#-------------------------------------------------------------------------------
#                          Analog Control Outputs
#-------------------------------------------------------------------------------
MFC   aout  0.000000

#-------------------------------------------------------------------------------
#                             PID Control Loops
#-------------------------------------------------------------------------------
heat1  loop  0.000000  off
heat2  loop  0.000000  off

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

4 Some Coding Examples

This section looks at some code snippets that show various aspects of the library in action. These are not meant to be stand-alone programs. In these examples, assume the library is using the example configuration file shown above unless otherwise specified.

This first example demonstrates how to instantiate a cache manager object and initialize it

with values from a configuration file. All of this is done with a call to the `CacheManager` constructor. Keep in mind that the cache manager requires client interfaces to various servers, and it is up to the user to create those interfaces and provide pointers to them for the cache manager.

```
#include "analogClientLib.h"
#include "digitalClientLib.h"
#include "loggerClientLib.h"
#include "outputChannelCache.h"
#include "pidClietnLib.h"
...

AnalogClient  *ac;
DigitalClient *dc;
LoggerClient  *lc;
PIDClient     *pc;

ac = new AnalogClient("example", ANADIG_SERVER_NAME);
dc = new DigitalClient("example", ANADIG_SERVER_NAME);
lc = new LoggerClient("example", LOGGER_SERVER_NAME);
pc = new PIDClient("example", PID_SERVER_NAME);

CacheManager cm(ac, dc, lc, pc, "ourConfig.cfg");
```

In the next example, assume that a `CacheManager` object called `cm` has already been instantiated, per the previous example. Here we use the simple "set" mechanism to set the values of two valves, a mass flow controller and a sample heater.

```
cm.SetDigital("v101", true);      // open this valve
cm.Setdigital("v300", false);     // close this valve
cm.SetAnalog("mfc", 120.0);       // set flow to 120 ml/min
cm.SetLoop("heat1", 50.0, true);  // enable the sample heater to maintain 50 degrees
```

When using the `Set...()` interface, any output channel that is already set to the specified value will *not* generate a client request to set its value. However, any unspecified output channels retain whatever values they had. They *are not* automatically set back to their default values.

In the final example, we set the same values on the same valves and so on, as we did in the previous example, only this time we want every other output channel managed by the library to be set automatically to its default value.

```
// First clear the new values group in preparation for defining a new
// set of channel value updates.
cm.ClearNewValues();

// Now define all the changes we want made.  Any channel that is NOT
// defined here will get set to its default value.
cm.AddDigital("v101", true);      // open this valve
cm.AddAnalog("mfc", 120.0);       // set flow to 120 ml/min
cm.AddLoop("heat1", 50.0, true);  // enable the sample heater to maintain 50 degrees
```

```
// Finally, make it happen!
cm.PushNewHardwareSettings();
```

Note in the above example that valve "v103" was not explicitly closed as it was in the `Set...()` example above it. That's because "off" (closed) is the default state for "v103." If it happens to be open at the time the code snippet runs, `PushNewHardwareSettings()` will make sure it gets closed.

Pacific Northwest
NATIONAL LABORATORY

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

U.S. DEPARTMENT OF
ENERGY

902 Battelle Boulevard
P.O. Box 999
Richland, WA 99352
1-888-375-PNNL (7665)
www.pnnl.gov