Technical Document **9/26/01** **LIGO-T010104-00-Z**

Defining and Testing Operational State Conditions in the Data Monitoring Tool

D. Chin (dwchin@umich.edu)
K. Riles (kriles@umich.edu)

University of Michigan Physics Department, Harrison Randall Laboratory
500 E. University Ave., Ann Arbor, MI 48109-1120

Abstract

This document is a user's guide to defining and testing Operational State Conditions (OSCs) in the Data Monitor Tool (DMT) background environment. It is assumed the reader knows the rudiments of creating a background monitor in the DMT using the `DatEnv` class.

Features

- Defining and monitoring of flexible conditions on data channels via start-time configuration files.
- Conditions can be defined w.r.t. thresholds on time series or power series data, using parameters specified in configuration files.
- Parameter values may also be specified via EPICS channels, giving real-time control.
- Boolean combinations of conditions with arbitrary complexity can also be defined.
- Optional parameters need not be specified in configuration file: default values are pre-defined.
- Configuration files allow an `include` directive to include standard pre-defined conditions, *e.g.* one can `include` the configuration that is currently used by a running monitor, such as `LockLoss`[1], and thus use the same conditions the monitor is using.
- Analysis code has access upon request to the real-time values being compared to thresholds that define conditions.
- The configuration files support inline comments.

Introduction

Before undertaking an analysis, one often needs to specify required interferometer or environment conditions. For example, before looking for instability in the Recycling Mirror (RM) servo[2], one would likely require both that servo and the Beam Splitter (BS) servo to be locked. One might also want Wave Front Sensing (WFS) to be engaged. In another analysis, one might require the laser intensity to exceed a threshold or require seismic motion to fall below a ceiling. To allow for flexible and comprehensive setting of conditions, one would also like to specify Boolean combinations of conditions. For example

```
Cond_1 = RM servo locked
Cond_2 = BS servo locked
Cond_3 = Cond_1 & Cond_2
```

The OSC tool has been written to support convenient defining and run-time checking of such conditions. Definitions can be made via a text configuration file or directly in DMT analysis code. A sampling of "standard" conditions (e.g., "`Arm_locked`" and "`LVEA_quiet`") have been provided in sample configuration files for analysis of engineering run data. It is expected that many more such standard conditions will be defined and refined as physicists gain more experience with his and future datasets. Definitions for useful conditions should be sent to the authors (`dwchin@umich.edu`, `kriles@umich.edu`) for incorporation into a public repository.

Quick Sample of Code

Before detailing how to use the OSC tool, let's first get a flavor of it with a stripped-down sample of code. The following code defines conditions and checks one of them in real time:

```
// During initialization:
osclist.readConfig("osc_sample.config");

// As each time interval of data (e.g., 1-second frame) is read:
if ( osclist.satisfied("LVEA_quiet") ) {
  // Do some analysis....
}
```

where the `LVEA_quiet` condition is defined by a line in the configuration file osc_sample.config (see below). One can also loop through *all* OSCs defined in a configuration file using an iterator:

```
// During initialization:
osclist.readConfig("osc_sample.config");

// As each time interval of data is read:
OperStateCondList::iterator iter = osclist.begin();
```

```
for (; iter != osclist.end(); ++iter) {
    if ((*iter).second->satisfied() == true)
        cout << "'" << (*iter).first << "'\tsatisfied" << endl;
    else
        cout << "'" << (*iter).first << "'\t\tNOT satisfied"
             << endl;
}
```

This scrap of code assumes that the monitor class has been written with `DatEnv` as a base class, and `osclist` is a member `OperStateCondList` object.

The `readConfig()` call is made in the monitor's constructor. The `satisfied()` call is made from the monitor's `ProcessData` method. The configuration file might look like:

```
# A comment
x_quiet    rmsrange "H0:PEM-LVEA_SEISX" lo=0. hi=2000.
y_quiet    rmsrange "H0:PEM-LVEA_SEISY" lo=0. hi=2000.
z_quiet    rmsrange "H0:PEM-LVEA_SEISZ" lo=0. hi=2000.
LVEA_quiet boolean  "x_quiet & y_quiet & z_quiet"   # a comment

# in general, lines look like
OscName  osc_type "channel_name" paramname=paramvalue  ...
```

The first three conditions have the names "`x_quiet`", "`y_quiet`" and "`z_quiet`". Each is of type `rmsrange` which means the data channel with the specified name must have an RMS value between the minimum and maximum values specified (0. and 2000. in each case). The fourth condition is Boolean, in this case the logical AND of the first three conditions.

Fields are separated by spaces or tabs. Valid condition names must begin with an alphabetic character and contain only alphanumeric characters or "_" (underscore). For `boolean` conditions, spaces are not required in the Boolean expression. The bitmasks can be specified in decimal, octal, and hexadecimal format: octal numbers are denoted by a leading "0" (zero) and hexadecimal numbers are denoted by a leading "0x".

Other Directives

There are other directives that can be specified in the configuration file which are not directly involved in defining OSCs.

`include`

- **Effect:** Similar to "`#include`" for the C preprocessor, it includes other configuration files in line. If the file is not found, a warning message is printed and the directive is ignored: processing continues.
- **Syntax:** There are two ways of using the `include` directive:
  - `include "otherFile.conf"` includes a configuration file in the same directory as the current file

– `include <otherFile.conf>` includes a configuration file from the directory given by the environment variable `DMTPARS`. If the environment variable is not set, a warning message is printed and the directive is ignored: processing continues.

`ignore`

- **Effect:** Deletes the named OSC from the `OperStateCondList`.
- **Syntax:** `ignore "previouslyDefinedOSCname"`

`debuglevel`

- **Effect:** Sets the debug level of the `OperStateCondList`.
- **Syntax:** `debuglevel 2` – the argument must be an integer. Only values ≥ 0 make sense, though negative values do not cause an error.

Conditions

The following list of OSC types are expected to expand with future releases of the OSC tool.

There are two types of OSCs: atomic OSCs, and meta-OSCs. The atomic OSCs specify conditions on the data stream, whereas the meta-OSCs specify conditions on other OSCs. Each OSC has some number of parameters which specify the details of the condition, *e.g.* threshold, frequency band, etc.

The atomic OSCs require a channel name in the configuration file definition, while the meta-OSCs require another OSC name in the definition. The OSC upon which a meta-OSC acts must have been defined before the meta-OSC.

Some classes of OSCs have parameters which others do not, *e.g.* OSCs which place conditions on each sample in the time-series data have a `fraction` parameter which specifies the minimum fraction of data that must satisfy the condition for the OSC to be satisfied. The `fraction` parameter would make sense for something like `valueabove` but not for something like `meanabove`.

Some parameters are *optional*, meaning that they need not be specified explicitly in the configuration file. If a parameter is optional it always has some default value.

If a line defining an OSC in the configuration file contains parameters that are not part of that OSC or meta-OSC, an undefined error may result.

Common Parameters

This is a list of names of parameters which are used for more than on OSC type. Note that ∞ is to be interpreted as the limit of the computer representation of the appropriate data type. A *stride* corresponds to one Frame of data. (FIXME)

`fraction`

- **Definition:** Specifies the fraction of the time-series data that must satisfy the condition for the OSC to be satisfied

- **OSCs:** `valueabove`, `valuebelow`, `valuerange`, `bitand`, `bitnand`, `bitor`, `bitnor`
- **Optional:** Yes
- **Data type:** Floating point
- **Legal values:** $(-\infty, 1]$ Any negative value means that the OSC is satisfied if at least one data point satisfies the condition
- **Default value:** -1

`hold`

- **Definition:** Specifies the number of additional strides for which to hold the `OSC.satisfied()` at True.
- **OSCs:** `transitup`, `transitdown`
- **Optional:** Yes
- **Data type:** Integer
- **Legal values:** $[0, \infty)$ The OSC will always be true for at least one (1) stride, and will be held True for an additional `hold` stride.
- **Default value:** 0

`dead`

- **Definition:** Specifies the number of strides beyond the end of a `hold` period for which the OSC cannot be True
- **OSCs:** `transitup`, `transitdown`
- **Optional:** Yes
- **Data type:** Integer
- **Legal values:** $[0, \infty)$ The OSC will be held False for `dead` strides
- **Default value:** 0

`threshold`

- **Definition:** Specifies an amplitude threshold for `*above`, and `*below`, and a change threshold for `*rise`, and `*fall` OSCs
- **OSCs:** `valueabove`, `valuebelow`, `meanabove`, `meanbelow`, `rmsabove`, `rmsbelow`, `abspowerabove`, `abspowerbelow`, `abspowerrise`, `abspowerfall`, `abspowergain`, `fractpowerabove`, `fractpowerbelow`
- **Optional:** No

- **Data type:** Floating point
- **Legal values:** $(-\infty, \infty)$
- **Default value:** N/A

`nstrides`

- **Definition:** Specifies the number of strides over which an average is to be computed
- **OSCs:** `abspowerrise`, `abspowerfall`, `abspowergain`, `meanrise`, `meanfall`
- **Optional:** Yes
- **Data type:** Integer
- **Legal values:** $[1, \infty)$
- **Default value:** 1

`lo`

- **Definition:** Specifies the lower limit for `*range` OSCs. OSC will be satisfied if `fraction` of data lies between lower and upper limits (see `hi`)
- **OSCs:** `valuerange`, `meanrange`, `rmsrange`, `fractpowerrange`
- **Optional:** No
- **Data type:** Floating point
- **Legal values:** $(-\infty, \infty)$
- **Default value:** N/A

`hi`

- **Definition:** Specifies the upper limit for `*range` OSCs. OSC will be satisfied if `fraction` of data lies between lower and upper limits (see `lo`)
- **OSCs:** `valuerange`, `meanrange`, `rmsrange`, `fractpowerrange`
- **Optional:** No
- **Data type:** Floating point
- **Legal values:** $(-\infty, \infty)$
- **Default value:** N/A

`freqlo`

- **Definition:** Specifies the lower frequency limit for OSCs which depend on the power spectrum.

- **OSCs:** `abspowerabove`, `abspowerbelow`, `abspowerrange`, `abspowerrise`, `abspowerfall`, `abspowergain`, `fractpowerabove`, `fractpowerbelow`, `fractpowerrange`
- **Optional:** No
- **Data type:** Floating point
- **Legal values:** $(-\infty, \infty)$
- **Default value:** N/A

`freqhi`

- **Definition:** Specifies the upper frequency limit for OSCs which depend on the power spectrum.
- **OSCs:** `abspowerabove`, `abspowerbelow`, `abspowerrange`, `abspowerrise`, `abspowerfall`, `abspowergain`, `fractpowerabove`, `fractpowerbelow`, `fractpowerrange`
- **Optional:** No
- **Data type:** Floating point
- **Legal values:** $(-\infty, \infty)$
- **Default value:** N/A

`mask`

- **Definition:** The bit mask with which to operate on the data.
- **OSCs:** `bitand`, `bitnand`, `bitor`, `bitnor`
- **Optional:** No
- **Data type:** Integer. (May be specified in octal or hexadecimal format: octal has a leading "0" (zero) and hexadecimal has a leading "0x" (zero x).)

Atomic OSCs

These OSCs act on the time-series data in channels, and hence require a channel name in their definition, *e.g.*:

```
X_arm_locked  meanabove "H2:LSC-AS_I" threshold=17.3
```

NOTE: The power in the `power*` OSCs is really band-limited RMS.

`valueabove`

- **True when:** `fraction` of data samples are greater than `threshold`
- **Parameters:** `fraction`, `threshold`

`valuebelow`

- **True when:** `fraction` of data samples are less than `threshold`
- **Parameters:** `fraction`, `threshold`

`valuerange`

- **True when:** `fraction` of data samples in the interval (lo, hi) and $<$ `hi`
- **Parameters:** `fraction`, `lo`, `hi`

`meanabove`

- **True when:** mean of data is greater than `threshold`
- **Parameters:** `threshold`

`meanbelow`

- **True when:** mean of data is less than `threshold`
- **Parameters:** `threshold`

`meanrange`

- **True when:** mean of data is in the interval (lo, hi)
- **Parameters:** `lo`, `hi`

`meanrise`

- **True when:** mean of data has increased by an amount ¿ `threshold` over `nstrides` strides
- **Parameters:** `threshold`, `nstrides`

`meanfall`

- **True when:** mean of data has decreased by an amount ¿ `threshold` over `nstrides` strides
- **Parameters:** `threshold`, `nstrides`

`rmsabove`

- **True when:** RMS (full bandwidth) of data is greater than `threshold`
- **Parameters:** `threshold`

`rmsbelow`

- **True when:** RMS (full bandwidth) of data is less than `threshold`
- **Parameters:** `threshold`

`rmsrange`

- **True when:** RMS (full bandwidth) of data is in the interval (lo, hi)
- **Parameters:** `lo`, `hi`

`bitand`

- **True when:** bitwise AND of `fraction` of data with `mask` is equal to `mask`
- **Parameters:** `mask`, `fraction`
- **Note:** `mask` may be specied in octal (leading `0`) or hexadecimal (leading `0x`), *e.g.* 0123 is decimal 83, 0xff is decimal 255

`bitnand`

- **True when:** bitwise AND of `fraction` of data with `mask` not equal to `mask`
- **Parameters:** `mask`, `fraction`
- **Note:** `mask` may be specied in octal (leading `0`) or hexadecimal (leading `0x`), *e.g.* 0123 is decimal 83, 0xff is decimal 255

`bitor`

- **True when:** bitwise AND of `fraction` of data with `mask` is not equal to 0
- **Parameters:** `mask`, `fraction`
- **Note:** `mask` may be specied in octal (leading `0`) or hexadecimal (leading `0x`), *e.g.* 0123 is decimal 83, 0xff is decimal 255

`bitnor`

- **True when:** bitwise AND of `fraction` of data with `mask` is equal to 0
- **Parameters:** `mask`, `fraction`
- **Note:** `mask` may be specied in octal (leading `0`) or hexadecimal (leading `0x`), *e.g.* 0123 is decimal 83, 0xff is decimal 255

`abspowerabove`

- **True when:** power in frequency range `freqlo` and `freqhi` is above `threshold`

- **Parameters:** `freqlo`, `freqhi`, `threshold`
- **Note:** For this condition and the other "power" conditions that follow, the computed power is normalized so that its sum from zero to the Nyquist frequency equals the mean square value of the corresponding time series. No windowing is performed.

`abspowerbelow`

- **True when:** power in frequency range `freqlo` and `freqhi` is below `threshold`
- **Parameters:** `freqlo`, `freqhi`, `threshold`

`abspowerrange`

- **True when:** power in frequency range `freqlo` and `freqhi` is between `lo` and `hi`
- **Parameters:** `freqlo`, `freqhi`, `lo`, `hi`

`abspowerrise`

- **True when:** power (RMS) in frequency range [`freqlo`, `freqhi`] has risen by an amount $>$ `threshold` over `nstrides` strides
- **Parameters:** `freqlo`, `freqhi`, `threshold`, `nstrides`

`abspowerfall`

- **True when:** power (RMS) in frequency range [`freqlo`, `freqhi`] has fallen by an amount $>$ `threshold` over `nstrides` strides
- **Parameters:** `freqlo`, `freqhi`, `threshold`, `nstrides`

`abspowergain`

- **True when:** Depends on value of `threshold`. If `threshold` > 1, True when power in frequency range [`freqlo`, `freqhi`] is changing by a factor $>$ `threshold` over `nstrides` strides. If `threshold` < 1, True when power in frequency range is changing by a factor $<$ `threshold` over `nstrides` strides.
- **Parameters:** `freqlo`, `freqhi`, `threshold`, `nstrides`

`fractpowerabove`

- **True when:** fractional power in frequency range [`freqlo`, `freqhi`] is above `threshold`
- **Parameters:** `freqlo`, `freqhi`, `threshold`

- **Note:** fractional power is defined to be the ratio of power in the frequency range requested to the full-band power

`fractpowerbelow`

- **True when:** fractional power in frequency range [`freqlo`, `freqhi`] is below `threshold`
- **Parameters:** `freqlo`, `freqhi`, `threshold`
- **Note:** fractional power is defined to be the ratio of power in the frequency range requested to the full-band power

`fractpowerrange`

- **True when:** fractional power in frequency range [`freqlo`, `freqhi`] is between `lo` and `hi`
- **Parameters:** `freqlo`, `freqhi`, `lo`, `hi`
- **Note:** fractional power is defined to be the ratio of power in the frequency range requested to the full-band power

`fractpowerrise`

- **True when:** fractional power in frequency range [`freqlo`, `freqhi`] has risen by an amount $>$ `threshold` over `nstrides` strides
- **Parameters:** `freqlo`, `freqhi`, `threshold`, `nstrides`
- **Note:** fractional power is defined to be the ratio of power in the frequency range requested to the full-band power

`fractpowerfall`

- **True when:** fractional power in frequency range [`freqlo`, `freqhi`] has fallen by an amount $>$ `threshold` over `nstrides` strides
- **Parameters:** `freqlo`, `freqhi`, `threshold`, `nstrides`
- **Note:** fractional power is defined to be the ratio of power in the frequency range requested to the full-band power

Meta-OSCs

`boolean`

- **True when:** Boolean expression of OSCs evaluates to True
- **Parameters:** none
- **Example:**`Both_arms_locked boolean "X_arm_locked & Y_arm_locked"`

- **Notes:** Boolean expressions use previously defined OSC names as Boolean variables. Operators supported are: `!` (NOT), `&` (AND), `|` (OR). Precedence rules follow those of standard logic. Parentheses may be used to make precedence explicit. (Ignore backslashes above in the HTML version of this manual.)

`transitup`

- **True when:** given OSC changes from False to True
- **Parameters:** `hold`, `dead`
- **Example:** `X_arm_lock_acquired transitup "X_arm_locked" hold=0`
- **Notes:** This meta-OSC is held True for `hold` + 1 strides

`transitdown`

- **True when:** given OSC changes from True to False
- **Parameters:** `hold`, `dead`
- **Example:** `X_arm_lock_lost transitdown "X_arm_locked" hold=0`
- **Notes:** This meta-OSC is held True for `hold` + 1 strides

Remarks on usage

The bitwise conditions merit further explanation. The `bitand` and `bitnand` conditions refer to whether *every* bit in the bitmask matches a bit in the data channel value. The `bitor` and `bitnor` conditions refer to whether *at least one* of the bits in the bitmask has a corresponding bit in the data channel value. Hence for bit masks with only one bit turned on, the `and` and `or` conditions are identical.

`transitup` and `transitdown` are conditions on other OSCs. `transitup` becomes True and is held True for $N + 1$ strides whenever the named OSC changes state from False to True, where N is the hold duration parameter. `transitdown` works in a similar way, except that it becomes True when the named OSC goes from True to False. The `dead` parameter prevents `transitup` and `transitdown` conditions from becoming True again for an interval of `dead` strides after the end of the `hold` period.

The various ways of specifying rises and falls in band-limited power are provided for flexibility during monitor developement. Very stable channels with nearly constant total power may be well suited to the abspower conditions. Channels with time-dependent total power but stable spectral shape may be well suited to fractpower conditions. Channels with large fluctuations in total power and in spectral shape may be best suited to conditions on powergain factors.

One cannot yet define OSCs in source code, only via a configuration file.

Checking conditions

If an OSC has been defined during initialization, then one can check whether it is satisfied during a given time interval with a call to the `satisfied()` method. *e.g.*:

```
if (osclist.satisfied("oscname"))
    // send a trigger
```

The `OperStateCondList` class inherits from a `hash_map<const string, osc::OperStateCond*, osc::hash<const string> >`, so all the methods that a `hash_map` has are available. (See SGI's documentation for the Standard Template Library at `http://www.sgi.com/tech/stl/`.) So, for instance, one can loop through all defined conditions:

```
// As each time interval of data is read:
OperStateCondList::iterator iter = osclist.begin();
for (; iter != osclist.end(); ++iter) {
    if ((*iter).second->satisfied() == true)
        cout << "'" << (*iter).first << "'\tsatisfied" << endl;
    else
        cout << "'" << (*iter).first << "'\t\tNOT satisfied"
             << endl;
}
```

Other Facilities

In addition to the `OperStateCondList` class, a helper class called `TSWindow` is also available. `TSWindow` represents a time series with at most N elements. It is typically used as a "window" on a time series that is to be presented to the DMT Viewer. For a relatively simple example, see the source code for the `LockLoss` monitor: `dmt/cvs/dmt/src/monitors/LockLoss`. To add a datapoint to a `TSWindow` object, use the `append()` method.

Hacker's Guide

Big picture from the end-user standpoint

We want to have various Operational State Conditions, encapsulated by the class `OperStateCond` (and abbreviated OSC for the rest of this document), which specify conditions which may or may not be satisfied by the data stream in *one (1)* channel.

Then, for the user, there is a `hash_map` of standard OSCs, *e.g.* `valueabove` and `meanbelow`, indexed by the user-specified names defined in a configuration file. This is so that the user can refer to specific OSCs.

Each OSC has none or some parameters associated with it, *e.g.* `valueabove` has two parameters, one of which is its `threshold`: if the data in the channel ever goes above that threshold value, the OSC is said to be "satisfied" (the `satisfied()` member function).

Each parameter has a name, *e.g.* `threshold`, and a datatype, *e.g.* `double`.

This code will read a configuration file specifying the various OSCs and the corresponding parameter values. Each OSC must be given a unique name, a string of alphanumeric characters beginning with an alphabetic character.

The reason for naming each OSC is so that each OSC maybe referred to later in a configuration file and used in meta-OSCs. Since each OSC.`satisfied()` returns a `boolean`, one may now define a Boolean OSC to say something like "True if CHAN1 has valueabove 2.3 AND CHAN2 has meanbelow 4.2".

Rather than having to manually type in the names of each OSC into the monitor code, we define the `OperStateCondList` class, which, despite its name, is not a linked-list but a `hash_map`, *i.e.* a dictionary that associates a pointer to an `OperStateCond` object with a string (the user-defined name of the OSC, specified in the configuration file). Making `OperStateCondList` a `hash_map` means that we may iterate over all defined OSCs:

```
OperStateCondList mOSClist;
OperStateCondList::iterator iter = mOSClist.begin();
for (; iter != mOSClist.end(); ++iter)
       if ((*iter).second->satisfied())
           // generate and send a trigger to the MetaDB
```

Notice that this still does not obviate the need for `OperStateCondList::satisfied(const char *oscname)` since the user might want to test for a particular OSC, and the method was defined in a previous incarnation of this code. Another way of accessing particular OSCs is to use the fact that it's a `hash_map` (which is actually what `OperStateCondList::satisfied()` does):

```
OperStateCondList mOSClist;
mOSClist.readConfig("configfile.conf");
```

```
if (mOSClist["my_condition"]->satisfied())
   cout << "my_condition is satisfied" << endl;
else
   cout << "my_condition is NOT satisfied" << endl;
```

Big picture from the programmer's standpoint

This is more info for one who wishes to modify the OperStateCondList code. If you read the above, you will see a very striking resemblance between OSCs and the basic datatypes of a programming language. In a programming language, we may define variables of different datatypes; in the OperStateCondList system, we may define OSCs of different OSC types (like `valueabove`).

And then, the Boolean OSCs are just a Boolean expression of previously defined OSCs. So, this really becomes a small language, and hence the scanning, tokenizing and parsing objects associated with the Boolean OSC. (See `osc/boolean/*.hh`.)

`boolean`, `transitup`, and `transitdown` OSCs are really meta-OSCs, *i.e.* they define conditions on other OSCs. So, to know whether the OSCs they refer to are satisfied, they need to get a pointer to the `OperStateCondList` that contains them.

To make the evaluation of the Boolean expression a little more efficient, we convert the infix form of the Boolean expression, *e.g.* `Locklost & PSLglitch`, to Reverse Polish Notation (RPN), and in the process verify the correctness of the expression. RPN is quicker to evaluate since it simplifies putting the running value in a stack.

To make evaluations for the other conditions a little more efficient, we store a short history of two `satisfied()` values: the value for the current stride, and the value for the previous stride. Some OSCs depend on more than just the previous stride, so they have their own private data members defined for storing this data.

Now, for a larger overview. And let's do this in a top-down way, with a very large grain.

We've already seen that `OperStateCondList` is a `hash_map<const string, OperStateCond*>`.

Any new types of OSCs that are to be written will have to inherit from `OperStateCond`: it is a base class that provides data and functions that all OSCs need.

`OperStateCondList` also contains ('has a' relationship) information about the various basic OSC types. This information is stored in `osc::TypeInfoMap mTypeInfoMap`. (See `OSCListMisc.hh`.)

`osc::typeInfoMap` is a `hash_map<const string, TypeInfo>`, *i.e.* it's a dictionary that relates a `TypeInfo` object to a `const string` (which is the name of that type, *e.g.* `valueabove`).

The `TypeInfo` class contains information about the OSC type. It does this by containing a sorted map of parameter types: `typedef map<const string, tseriesType_t> ParamTypeMap`. `tseriesType_t` is defined in `namespace osc`

and is an enumeration of the various "atomic" data types that may be contained in a `TSeries`. The indexing key for `ParamTypeMap` is the name of the parameter, *e.g.* `mParamTypeMap["threshold"]` would be a `DOUBLE`.

In summary, let "*" mean "has a", and ">" mean "is a".

```
OperStateCondList
 > hash_map of OperStateCond*, indexed by user-defined names of OSCs
 * TypeInfoMap, containing information about the basic types

OperStateCond
 > a base class from which actual OSCs will inherit
 * a bunch of stuff.  See OperStateCond.hh and misc.hh

TypeInfoMap
 > hash_map of TypeInfo, indexed by names of types
 * ParamInfoMap, containing information about the parameters
                 associated with this type of OSC

ParamInfoMap
 > hash_map of ParamInfo, indexed by names of parameters
```

The `ParamInfoMap` and `TypeInfoMap` are initialized in `OperStateCondList`'s constructor. Since this data should be common to all instances of `OperStateCondList`, one would like this to be a static data member, but unfortunately it is almost impossible to initialize a complex data structure such as this at compile time.

Now, how do the OSCs store the values of their parameters? Since these parameters may be of different types, and they may be read from an Epics channel, we can't just store them as simple values. We use a template class (just because I don't know how to do something like the `Param` class in DMT: see `osc/Param.hh`). For now, we only have `int` and `double` parameters, so each `OperStateCond` object has two `hash_maps`: one for integer parameters (`mIntParams`) and one for doubles (`mDblParams`). These two `hash_maps` are indexed by the names of the parameters. The value of a parameter may be retrieved by, *e.g.*:

```
mDblParams["threshold"].value()
```

How to Add New OSC Types

Look in the `osc` subdirectory. You'll create a new class that inherits from `OperStateCond`. Look at the others for examples. You'll have to initialize the object's `ParamInfoMap` in the `OperStateCondList` constructor, and then enter that into `OperStateCondList`'s `TypeInfoMap`. You will need to add a section that actually adds a new OSC to the `hash_map` (around line no. 612 and onwards). You may need to write sections to parse the config file line for your new OSC, though it's quite unlikely unless you have a new type of

parameter (*e.g* a complex number) that behaves very differently from the types that already exist. `OperStateCondList::parseAtomicParams()` is the method that parses config file lines for OSCs. The `transit*` conditions have config line parsing written inline in the constructor of `OperStateCondList`. Only the `boolean` OSC type has it's own parser since it's a bit of a trick to parse a Boolean expression.

References

[1] D. Chin and K. Riles, "Description of the DMT LockLoss Monitor", LIGO-T-010105-00-Z (September 2001).

[2] D. Chin and K. Riles, "Description of the DMT ServoMon Monitor", LIGO-T-010106-00-Z (September 2001).