CryoCore

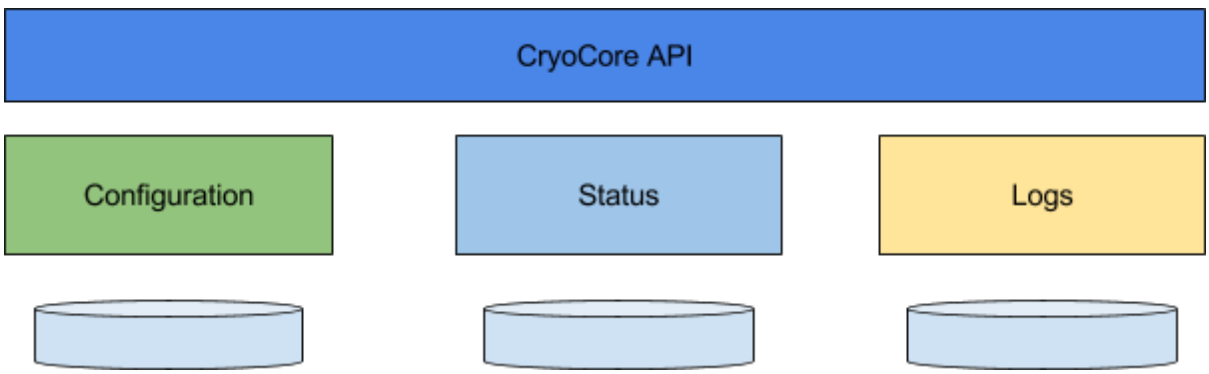
Documentation of APIs and services

Overview, Python

CryoCore is based on three different services: Configuration, Logs and Status. All three are stored in databases (MySQL), and the databases can either be the same one, or different databases. For example, configuration can be local, while status info be stored on a remote host.

In a nutshell:

```
import CryoCore
try:
    cfg = API.get_config("MyModule")
    log = API.get_log("MyModule")
    status = API.get_status("MyModule")
    log.info("My module version %s starting" % cfg["version"])
    status["state"] = "Initializing"
    status["processed"] = 0
    ...
finally:
    CryoCore.API.shutdown()
```



The CryoCore API provides a simple entry point to the CryoCore. Four functions are normally used:

- get_config(root) - get a Configuration object for the given software module
- get_status(channel) - get a Status object for the software module
- get_log(module_name) - get a log object for the software module
- shutdown() - Shut down all CryoCore services

Note that API.shutdown() should be called when your program exists, as several services are asynchronous and should be allowed to shutdown cleanly, flushing values to databases etc. The API also provides an event - API.api_stop_event that can be monitored by your applications - when shutdown() is called, the api_stop_event is set.

Configuration is a hierarchically structured tree of parameters that can be updated dynamically on runtime. In general, any magic number should be a configuration parameter. As config parameters can change at runtime, loops that require a stable value should store it in a local variable while looping.

Status provides a simple way to store status information. Parameters describing the running state of your software should be status elements. Examples of state can be percent done, “running”, “idle”, number of pictures taken etc. Note that state elements hold the values during runtime, so there is normally no need to also store the value as a variable.

Log is based on the python “logging” system. Log messages will be stored in a database, including file names, line numbers etc.

All three services use databases, possibly the same one (can be configured). Inserts for status and logs are asynchronous, and thus nonblocking. All entries are timestamped (using the local clock - NTP or equivalent is suggested, in particular on production systems). All software should run on both Python2 and Python3.

Installation

CryoCore is available in subversion from trac.itek.norut.no. Check out with:

```
svn co svn+ssh://trac.itek.norut.no/home/subversion/cryocore
```

If you don’t have CryoCore installed already, run the install.sh script located in CryoCore/Install/. If you use Ubuntu, this should be sufficient. If you use some other distro or OS, please manually install mysql and the mysql-connector.

The process will ask you for passwords - one is to allow you to become root to install packages (only for Ubuntu). The other password requested is the mysql root user password.

We suggest installing argcomplete to enable autocomplete. This includes running activate-global-python-argcomplete. The install script installs these automatically on Ubuntu.

Configuration

```
import CryoCore
cfg = CryoCore.get_config("Instruments.Camera")

# Set 'enabled' to true as default. If already set, this is ignored
cfg.set_default("enabled", True)

# We require the parameter 'shutter_speed', but do not create it.
# If possible, rather use a good default value.
cfg.require(["shutter_speed"])

# Define a callback function for configuration changes
```

```
def onShutterSpeedChange(param):
    somecamobj.set_shutter_speed(param.get_value())

# Register the callback function
cfg.add_callback("shutter_speed", onShutterSpeedChange)

while CryoCore.API.api_stop_event.isSet():
    if cfg["enabled"]:
        # Do something here
    else:
        # Do something else
```

Versions:
The configuration has support for multiple versions. There is a version “default”, and that configuration also has a “default_version” parameter (“default” is default). The ConfigTool can be used to manage multiple configurations.

API:

```
cfg = CryoCore.API.get_config(root, version)  # if root is not provided, it's the root
of the config.

# Get parameters
configParameter = cfg.get("parameter")  # Throws NoSuchParameterException if not found
value = cfg["parameter"]  # Returns None if not found,

# List sub-parameters (both parameters and folders)
a_list = cfg.keys()

# Versions
versions = cfg.list_versions()

# Copy a configuration from source_version to target_version - fails if parameters exist
and overwrite is False
cfg.copy_configuration(source_version, target_version, overwrite=False)

# Config Parameter objects have some methods too
fullname = configParam.get_full_path()
configParam.set_value(value, datatype=None, check=True, commit=True)
```

ConfigTool.py

In CryoCore/Tools/. Allows command line access to configuration. If argcomplete is installed properly and you use bash, autocomplete is available. This is highly recommended.

ConfigTool does provide some help, and the autocomplete makes it easier. A few features that are not easy to see:

- If ConfigTool.py is called with something that is neither a command, it will list the parameter requested, or if no such parameter exists, any parameter that contains the given keyword.

- A quick-hand is allowed, where ConfigTool.py Module.Parameter=value results in ‘value’ being set. The quick-hand checks type too, so setting an int to “True” for example will fail. Use “set” if you want to overwrite the type.

Example of serializing/deserializing:

```
# Export System.SystemControl tree, e.g. System.SystemControl.default_user
# becomes SystemControl.default_user in the somefile.json file
./CryoCore/Tools/ConfigTool.py serialize System.SystemControl > somefile.json

# Deserialize the whole tree under SomeRoot - e.g. SystemControl.default_user
# becomes SomeRoot.SystemControl.default_user
./CryoCore/Tools/ConfigTool.py deserialize SomeRoot somefile.json
```

ConfigUI.py

ConfigUI is a curses based configuration tool - it allows navigation using arrows, light editing etc. Parameters cannot be added or deleted

Status

```
import CryoCore

status = CryoCore.API.get_status("MyModule")
status["state"] = "Initializing"
status["processed_blocks"] = 0
while not CryoCore.API.api_stop_event.is_set():
    status["state"] = "processing"  # Only has an effect if changed
    # Process a block
    status["processed_blocks"].inc()
    if status["processed_blocks"].get_value() > MAX_BLOCKS_PROCESSED:
        break  # Should not process more blocks now
status["state"] = "idle"
```

Status objects can be thought of as local variables, that are also stored for later use. From your code, you can both read and write the value, but the value is only stored to the database, never retrieved. In other words, you cannot store a value in a status element for later retrieval. Use either config or some more suitable storage for that.

```
# Create status item
# Implicit creation:
status["item"] = value

 # Explicit creation
new(name, initial_value=None, expire_time=None)
new2d(name, size, initial_value=None, expire_time=None)
```

There are two ways to create status elements, explicit and implicit. Implicit status element creation, like **status[“someparam”] = somevalue** creates a single value status element with no expire time. Explicit creation is done using **new(name, initial_value=None, expire_time=None)** or **new2d(name, size, initial_value=None, expire_time=None)**. For a 2d status element, size is a tuple (int, int) where both numbers must be larger than 0. Expire time is in seconds, and is only for data storage Even if the value has expired, the status element will retain the value. Typical use for expire time is to visualize progress in a GUI, debugging etc.

```
# Set a value
status["item"] = value

status["item"].set_value(value, force_update=False, timestamp=None)
```

Note that a status parameter is only updated if it was changed. If you want to force an update, use **set_value(val, force_update=True)**. Timestamps are also implicitly “now”, if you have a timestamp for the status update already, use **set_value(val, timestamp=your_timestamp)**” where your_timestamp is an epoch value

Status elements can have any value (limited to 128 bytes), 2D status elements must be an integer between 0 and 255.

Getting the value of a status element should be done using get_value() to ensure that you get the correct data type. Operators have been overloaded to ensure that ==, !=, <, >, >= and <= work on the given values. Note that “0” != 0, etc.

You can also hook up to status updates to get events triggered on changes or on particular values. This can be done across threads, but not across processes.

```
import CryoCore
status = CryoCore.API.get_status("MyModule")
status["state"] = "initializing"

# Typically in another thread
eventC = threading.Event()
status["state"].add_event_on_change(eventC, once=True)
eventC.wait(5.0)  # Wait for up to 5 seconds for the event to trigger
if eventC.isSet():
   print("State changed")  #  It changed once, will not be triggered any more

# In a third thread
eventV = threading.Event()
status["state"].add_event_on_value("idle", eventV)
while not CryoCore.API.api_stop_event.isSet:
    while not eventV.isSet():
       eventV.wait(1)
    if event.isSet():
       print("Idling")
       eventV.clear()  # Prepare for another

# In the main thread (or any thread really)
status["state"] = "working"  # Will trigger eventC, "State changed" is printed
...
status["state"] = "idle"  # Will trigger eventV, "Idling" is printed
...
status["state"] = "working"  # Triggers nothing
...
status["state"] = "idle"  # Will trigger eventV, "Idling" is printed
```

TailStatus.py

TailStatus.py is a tool to view and tail status messages. It has autocomplete if you installed and configured argcomplete properly.

```
cryocore/trunk$ ./CryoCore/Tools/TailStatus.py -f
Tue Aug  2 11:44:23 2016 [UnitTest] runner4 = 98
Tue Aug  2 11:44:23 2016 [UnitTest] runner5 = 93
```

TailStatus.py -h gives usage. The channel in the above example is “MyModule”. If one or more full status names (channel:name - “MyModule:runner4 MyModule:runner5” to get the two in the example above), only status data for these modules are shown.

Logs

Logs are based on the Python logger, so it should be familiar to anyone that's used it. By default it stores its logs in a database, but it is possible (with a bit of tinkering) to add local files as a destination too. A log viewer is available as "TailLog.py".

Usage:

```
import CryoCore

log = CryoCore.API.get_log("MyModule")
log.debug("a debug message")
log.info("Information message")
log.warning("Something slightly bad happened")
log.error("Something very bad happened")
log.fatal("Something incredibly bad, like, insanely bad, occurred")
try:
    ...
except Exception:
    log.exception("I got an exception")
```

log.exception also logs the stack trace, so you don't need to pass the exception or create any fancy strings. Files, line numbers, levels, logger and timestamps are all collected.

TailLog.py

TailLog is a tool to view logs from a database. It also has autocomplete if installed correctly.

```
cryocore/trunk$ ./CryoCore/Tools/TailLog.py
Tue Aug  2 11:42:16 2016 [DEBUG][InternalDB( 294)][MySQLStatusReporter]Initializing tables
Tue Aug  2 11:42:17 2016 [DEBUG][InternalDB( 309)][MySQLStatusReporter]Initializing tables DONE
Tue Aug  2 11:42:21 2016 [ INFO][    Status( 132)][UnitTest]Callback UnitTest thread started
```

Confusingly, the logs have both modules and loggers. The module is the file name of the log statement (and the line number is in paranthesis in the statements). The logger is the name given to API.get_log(thisIsTheLoggerName).

Similarly to TailStatus, the logger can filter results. If you want to filter a logger or a module, use --logger <loggername> or --module <modulename>, if you just want to filter on keywords, just write them in a space separated list.