c699util Developer's Guide

If you have not already done so, please read the c699util User's Guide to familiarize yourself with c699util and its capabilities.

This article demonstrates how to add features to c699util and describes parts of the c699util build, test, and release process. I found it challenging to create c699util due to the Qt4, 699 C++ code, SWIG, and qmake learning curves, and so I hope to ease that learning process for others. I assume that readers are familiar with Qt4 and the 699 C++ code.

SWIG

SWIG (Simplified Wrapper Interface Generator) is the program used to add a Python interface (i.e., c699util) to the 699 C++ code. As of this writing, c699util is created using SWIG 3.0.8.

Let's assume that we have code in files called example.cpp and example.hpp, and that we want to use SWIG to make this code available in Python. Our next step would be to create an *interface file* called example.i, and then put the declarations of the code that we want to expose to Python in that file; this is mostly just copying and pasting the declarations in example.hpp. We could then run SWIG as follows:

```
swig -Wall -c++ -python example.i
```

and SWIG would generate two files: example_wrap.cxx and example.py. example_wrap.cxx now needs to be compiled into a shared object library, and example.py will look for that shared object library when it is imported. This means that example.py is useless until example_wrap.cxx is built.

We would then build example_wrap.cxx as a dynamic library and name the resulting file _example.so (SWIG expects this naming convention -- an underscore followed by the module name. Also, there's nothing wrong with renaming a dynamic library file. These file types have no strictly enforced file extension.).

At this point, we could open up our Python interpreter and type "import example". example.py would be imported, and it would then load _example.so. example.*pp now has a Python interface.

A key takeaway from this description is that "running SWIG" successfully meant generating two files -- example.py and _example.so. Together, these two files are our new Python module.

Of course, the devil is in the details, and it's easy to make mistakes in this process. So, developers who modify c699util (or create something similar to it) are encouraged to start by making sure that they can follow this tutorial (I had to get rid of the -arch i386 flag when I went through the article), which can be thought of as the "Hello, World!" of SWIG. As with any complicated software development, a smart approach is to start with something simple that works well (such as the example in this article) and then to make small changes/enhancements to it, ensuring that the small changes work as expected.

I found the SWIG documentation and mailing list to be very helpful, particularly:

- SWIG Documentation Prefix
  - SWIG Documentation Introduction

    - SWIG Basics
- SWIG and C++
- SWIG and Python

While those last two links lead to very long articles, I found it worthwhile to grit my teeth and read them in their entireties. If you find yourself maintaining c699util, you should probably do the same. However, if you're just going to make a small change to c699util, you can probably get away with just "cookbooking" off of the simpler examples in the first two links as well as existing c699util code. (If you do so, be sure to update test_c699util.py! This will be described below.)

Additionally, my personal notes on SWIG and c699util have been shared with the MOMA software team. Contact a team member if you need access.

How c699util and the 699 C++ code work together

c699util is defined in three files: c699util.cpp, c699util.hpp, and c699util.i. c699util.cpp and c699util.hpp (occasionally referred to as c699util.*pp) can be thought of as the "bridge" between the 699 C++ code and the Python scripts that want to use the features defined in the 699 C++ code. As such, c699util.*pp are responsible for hiding 699 C++ implementation details from those Python scripts. c699util.i contains the routines from c699util.*pp that we want exposed to Python; again, this is mainly done by copying declarations from c699util.hpp into c699util.i.

The general approach taken by c699util.*pp is to define a class that Python scripts expect, such as TMFile, and within that class, use 699 C++ objects to implement useful methods. For example:

```
class TMFile
{
public:
    // todo - I'm forgetting to use const in these methods
    TMFile(std::string filename);
    std::string filename();
    std::string absolute_directory();
    std::string directory_string();
    std::string get_mcal_filename();
    long long int file_size();
    int tid();
    int length();
    bool exists();
    TMPacket get_pkt(int index);

    double t0_in_unix_time() const;
```

```
private:
    QString filename_;
    TmMeta tmMeta_;
    MomGse::MomTelemetryFilePtr momaTlmFilePtr_;
    MomGse::MarkerCache markerCache_;
    MomGse::MomaTimestampResolver& timestampResolver_;
};
```

As you might expect, TMFile generates packet objects via momaTlmFilePtr_, and determines their marker and timestamp properties by using markerCache_ and timestampResolver_. TMPacket, ScienceDataCacheWrapper, and PythonStyleMomaScan follow the same design pattern.

How SWIG and qmake work together

I don't want to describe our entire build system, but there are a few c699util.pro details worth understanding:

```
macx {
    INCLUDEPATH += /Library/Frameworks/Python.framework/Versions/3.4/include/python3.4m
    LIBS +=        /Library/Frameworks/Python.framework/Versions/3.4/lib/libpython3.4.dylib
    INCLUDEPATH += /Library/Frameworks/QtCore.framework/Versions/4/Headers
}
linux-g++ {
    # When Python3 is installed on wherever, it should be installed with
    # "./configure --enable-shared". This is necessary so that the library files
    # generated contain position independent (-fPIC) code.
    INCLUDEPATH += /usr/local/include/python3.4m
    LIBS += /usr/local/lib/python3.4/config-3.4m/libpython3.4m.a
}
```

This section is the most likely to cause issues. Python can be installed in a few different ways, and in some of those ways these locations may not be valid. However, if developers install Python 3.4.3 from the Python website , then there shouldn't be any problems. Don't worry about the linux-g++ section for now. It's only for the CentOS virtual machine which is used to build c699util for mine699.

`system(swig -Wall -c++ -python c699util.i)`

Running qmake on the c699util.pro file invokes SWIG.

```
macx {
    QMAKE_LFLAGS_PLUGIN -= -dynamiclib
    QMAKE_LFLAGS_PLUGIN += -bundle
}
```

```
linux-g++ {
   QMAKE_LFLAGS_PLUGIN += -fPIC
}
```

When linking on OSX, replace the -dynamiclib flag with the -bundle flag, ensuring that c699util is in the Mach-O file format. I don't remember why this is necessary. When linking on the CentOS VM, produce position-independent code.

```
# Bundles have no strictly enforced file extension, but swig expects the
# libraries it uses to be of the form _<module name>{=html}.so , so rename the output
# file. (Don't forget the leading underscore. It's a common pitfall.)
macx {
   QMAKE_POST_LINK = "mv libc699util.dylib _c699util.so"
}
linux-g++ {
   QMAKE_POST_LINK = "mv libc699util.so _c699util.so"
}
```

That comment should sufficiently explain this part of c699util.pro.

How to add a feature to c699util

Getting c699util Source Code

c699util is located in svn://repos699.gsfc.nasa.gov/cppcode/qt4/trunk/c699util.

Setting up test_c699util.py

c699util/test_c699util.py is a fully automated test of c699util. I've done my best to ensure that every c699util feature is thoroughly tested inside of test_c699util.py.

So, if you find yourself modifying c699util, ***ensure that test_c699util.py remains a thorough test of c699util by adding good tests to it!*** This test suite runs on both OSX and CentOS and has saved us a lot of trouble by finding subtle bugs before our users could be affected by them. It should be run after every new feature is added to ensure that nothing else has broken (it runs in less than 60 seconds). Now, to set up your environment to run test_c699util.py, do the following:

If you have not done so, follow the instructions here to get MOMA Data View and check out MOMA data.

Now, if you have not already done so, tunnel into MOMAIOC and then open up a new tab. In this new tab, type

`cd ~`

`svn co svn://localhost:6994/momadata/test_data ~/momadata/test_data`

Then, open up any TID inside of test_data using MOMA Data View. This should add a MOMA_TEST_DATA group to your ~/.699config.INI file. Once you've done this, close MOMA Data View and open up ~/.699config.INI in a text editor. Modify this line:

```
[MOMA_TEST_DATA]
<other lines>
tm_database=/Users/mpallone/momagse/TMDef/MOM_TM_Database.txt
```

by putting "Test_" in front of MOM_TM_Database.txt. It should now look like:

```
[MOMA_TEST_DATA]
<other lines>
tm_database=/Users/mpallone/momagse/TMDef/Test_MOM_TM_Database.txt
```

This file is a copy of MOM_TM_Database.txt which has been frozen just for the sake of test_c699util.py. This way, future conversion changes to the real database won't break test_c699util.py.

Next, ensure that tmread is installed on your system.

At this point, you should have the necessary test_c699util.py dependencies. Building and running test_c699util.py is described in the next section.

Building c699util

Whenever I release a new version of c699util, I run the following commands:

```
make clean
qmake -spec macx-g++ c699util.pro # For CentOS, I can just run "qmake"
make release
cat top_of_c699util_dot_py.py c699util.py > temp_c699util.py && mv temp_c699util.py c699util.py
./test_c699util.py
./copy_into_699util.py # Only do this if test_c699util.py passes all tests!
```

(Actually, these commands are inside of the files osx_release and centos_release, and I just run one of those scripts depending on the OS I'm using.)

copy_into_699util.py is a special script which writes c699util metadata to a file called VersionInfo.txt, and then moves _c699util.so, c699util.py, and VersionInfo into their "release homes" in the 699util repository. (How c699util fits into the 699util release scheme is described later in this article; do not concern yourself with such details at this moment.) VersionInfo.txt is important because it tells software engineers who are debugging c699util (a) what version of c699util is running on a user's computer, and (b) if any uncommitted changes in the working copy made it into the user's version of c699util. Please only release c699util by using copy_into_699util.py.

Example Feature

Let's pretend that we want to add a description() method to c699util's TMPacket class. While we're at it, we'd like to make it so that calling str(pkt_object) in a Python script calls this description() method.

In c699util.hpp, we'd add the following:

```
/*****************************************************************************
 * TMPacket definition
 *****************************************************************************/
class TMPacket
{
public:
   TMPacket(GseLib::pbytes packetBytesPtr, int pktIndex,
            const MomGse::MarkerCache* markerCachePtr,
            const MomGse::MomaTimestampResolver& timestampResolver);

   std::string description();  // this method was just added
```

Next, in c699util.cpp, we would define the description() method:

```
/*****************************************************************************
 * TMPacket Implementation
 *****************************************************************************/

TMPacket::TMPacket(GseLib::pbytes packetBytesPtr, int pktIndex,
                   const MomGse::MarkerCache* markerCachePtr,
                   const MomGse::MomaTimestampResolver& timestampResolver)
   : packetBytesPtr_(packetBytesPtr),
     pktIndex_(pktIndex),
     markerData_(markerCachePtr->getMarker(pktIndex, &hasMarkerData_)),
     unixTimestamp_(timestampResolver.getPktUnixTime(pktIndex)),
     relativeTimestamp_(timestampResolver.getPktRelativeTime(pktIndex))
{
   // no further initialization needed
}

std::string description()
{
   return std::string("this is a description of a packet);
}
```

Then, in c699util.i, we would copy this declaration:

```
/*************************************************************************
 * TMPacket definition
 *************************************************************************/
class TMPacket
{
public:
    TMPacket(GseLib::pbytes packetBytesPtr, int pktIndex,
             const MomGse::MarkerCache* markerCachePtr,
             const MomGse::MomaTimestampResolver& timestampResolver);

    std::string description(); // this method was just added
    %pythoncode %{
        def __str__(self):
            return self.description()
    %}
```

Notice that the change we made to c699util.i is exactly the same as the change we just made to c699util.hpp, except for the `%pythoncode` stuff. (If we didn't want to add a __str__() method to TMPacket, then c699util.hpp and c699util.i would change in exactly the same way.)

`%pythoncode` is a SWIG feature that allows us to define Python code that appears in the Python file that the 'swig' command outputs (which, in our case, is c699util.py). The __str__() method defined above will now be a method of the TMPacket class, just as description() is a method of the TMPacket class. Once c699util is built, calling str(pkt_object) will invoke the __str__() method defined above, which will then call description(). %pythoncode is an extremely useful SWIG feature because it (a) allows us to use magical Python methods (such as generators and iterators) that are difficult or unrealistic to implement in C++, (b) allows us to use concise, powerful Python syntax when defining our new classes rather than writing needlessly verbose C++ code, and (c) allows us to import and use Python modules at runtime -- even tmread modules! For example, consider the following uses of %pythoncode in the TMFile class:

```
class TMFile

    <other TMFile methods go here>{=html}

    %pythoncode %{
        def test_name(self):
            return self.directory_string().split('-', maxsplit=5)[5]
    %}

    %pythoncode %{
        def message_log(self):
            from tmread import MessageLog
```

```
        return MessageLog(self)
    %}
```

Defining `test_name()` using `%pythoncode` lets take advantage of Python's convenient string manipulation features. Defining `message_log()` using `%pythoncode` makes it possible to use tmread's well-tested and powerful MessageLog class, because we can import and use it at runtime.

Release Process

This process should be followed every time c699util is released. It requires a previously mentioned CentOS virtual machine, which exists solely to build and test c699util for mine699. Passwords for this VM have been distributed to the MOMA team. Copies of the VM exist on Mark Pallone's external hard drive and on the MOMA Data iMac located (as of this writing) in GSFC Building 33 F109A, ECN #2377998, inside of /Users/lab/Documents/c699util-virtual-machines.

c699util Release Process

1. **Update test_c699util.py so that it thoroughly tests your new feature.**
2. Commit the code being released.
3. On your Mac, run ./osx_release. This will:
    1. do a clean build of c699util
    2. run test_c699util.py
    3. copy the relevant files (including working copy differences) into ~/labcode/699util/c699util, but only if the test succeeded.
4. cd into ~/labcode/699util/c699util/osx, and 'svn commit' the newly built files
5. On the CentOS build VM, "svn up" the momagse and qt4 directories.
6. Run ./centos_release. This will:
    1. do a clean build of c699util
    2. run test_c699util.py
    3. copy the relevant files (including working copy differences) into ~/labcode/699util/c699util, but only if the test succeeded.
7. cd into ~/labcode/699util/c699util/centos, and 'svn commit' the newly built files
8. Log into mine699. On your own account (e.g., 'mpallone' instead of the moma user):
    1. svn update the Python tools
    2. svn update ~/labcode/qt4
    3. cd into ~/labcode/qt4/c699util, and run test_c699util.py
    4. If this works, switch to the 'moma' user and update the Python tools
9. Build a new Python tools package, being sure to test that the package works on our OSX VMs.
10. Add a new entry to the release page.
11. Commit the new DMG to `momagse/Apps/py699`.
12. Document any new c699util features in the c699util user's guide.
13. If necessary, update and release `extract_hk_data_main` on mine699 as well. (Strictly speaking this has nothing to do with c699util, but we shouldn't forget about this script when we release updates to c699util.)

It's annoying to run the test in three places, but sometimes bugs exist in one environment but not others.

c699util dependencies for both OS X and CentOS

It's useful to be aware of c699util dependencies when debugging. On OSX, _c699util.so contains the following dependencies:

```
(py699) gs699-mpallone:c699util mpallone$ otool -L _c699util.so
_c699util.so:
	/Library/Frameworks/Python.framework/Versions/3.4/Python (compatibility version 3.4.0, current version 3.4.0)
	QtGui.framework/Versions/4/QtGui (compatibility version 4.8.0, current version 4.8.4)
	QtCore.framework/Versions/4/QtCore (compatibility version 4.8.0, current version 4.8.4)
	QtNetwork.framework/Versions/4/QtNetwork (compatibility version 4.8.0, current version 4.8.4)
	/usr/lib/libstdc++.6.dylib (compatibility version 7.0.0, current version 104.1.0)
	/usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1226.10.1)
	/usr/lib/libgcc_s.1.dylib (compatibility version 1.0.0, current version 915.0.0)
```

The expected location of these dependencies can be tweaked by using the install_name_tool program. Python 3.4 should be in the specified location if it is installed through the DMG file available on the Python website. The use of Homebrew in this particular case is discouraged. The Qt dependencies should be discoverable once Qt4 is installed. The remaining dependencies were already available on my Mac; I suspect that they're standard on all modern Macs.

Unlike the OSX distribution of c699util, the CentOS distribution has the Python and Qt dependencies statically compiled into the _c699util.so file. The .so file's remaining dependencies are all standard (i.e., they're available by default on CentOS machines such as mine699.gsfc.nasa.gov).

c699util, the 699 developer setup, and the 699util package release scheme

Releasing c699util means committing new versions of

- c699util.py
- _c699util.so
- VersionInfo.txt

into the repos699 SVN server, in the locations `labcode/699util/c699util/osx` and `labcode/699util/c699util/centos`. The rest of this section describes how c699util fits into the the 699util developer's setup and the 699util package release system.

Review of the 699util Environment

Even for those who are familiar with the 699util installation article, how all the different parts of 699util fit together can still be hard to understand. So, reviewing this should make the rest of this section much easier to understand.

The 699util environment has four major parts:

1. 699 Data
2. 699 tool configuration/settings
3. 699 Code

4. Local environment configuration

1. 699 Data. Setting up 699 data simply means checking out folders such as `momadata/etu`, `momadata/fm`, etc. from momaioc.gsfc.nasa.gov's SVN repository. Typically, momadata is checked out into the home directory, but this is not required.

2. 699 tool configuration/settings. Items in this category mainly tell 699 software where data is, and how to interpret that data. This category includes:

- checking out momagse
- using MOMA Data View to initialize the ~/.699config.INI file. Simply open up a TID for each model that you have checked out in your `momadata` working copy.
- (If you're doing a developer install, you will also have to checkout config699.)

3. 699 code. Developers can check out 699util from SVN. Users can find MOMA script DMGs here.

4. Local environment configuration. This tends to be the most opaque. It consists of the following:

- Installing Python 3.4.3 from the Python website, and updating your .bashrc, .bash_profile, or .profile file so that the newly installed version of Python can be found via the PATH environment variable.
- setting up a virtual environment using the program virtualenv. A virtual environment can be thought of as a folder containing libraries and executable programs, as well as the environment variables that reference those libraries and executables. In the 699util world, virtualenv is used to create either `~/py699` or `~/py699-moma`. Python 3.4.3 is copied into this directory and, in the case of the package install, 699util scripts are also copied. Typically, the last line of a user's `~/.bashrc` file "activates" the virtual environment, which essentially just puts the virtual environment's binary folder (e.g., `~/py699/bin`) at the front of the PATH variable. Then, when a scripts starts with `#!/usr/bin/env python3`, the first instance of the python3 executable found by the `/usr/bin/env` program will be the one inside of the `~/py699` directory.
- The location of the 699util scripts, such as tmplot.py. In the package install meant for users (i.e., non-developers), these scripts are simply copied into the virtual environment folder `~/py699/bin`. For developers who already have these scripts checked out into a working copy, the location of those scripts is added to the PATH variable as follows:
  - the developer opens up a terminal
  - the terminal sources the developer's `~/.bashrc` file
  - the `~/.bashrc` file sources the `~/config699/bash_config.sh` file, which adds the script directories to the PATH variable.
- The location of 699util libraries. This is very similar to how scripts are handled. For non-developers, the 699util Python library is in the virtual environment's library folder. For developers, the `~/config699/bash_config.sh` directory adds the libraries to the PYTHONPATH environment variable, which Python uses when looking for modules that need to be imported.

Developer Setup

As mentioned above, the developer install sources `~/config699/bash_config.sh`, which sets PATH and PYTHONPATH environment variables. `bash_config.sh` determines the version of c699util that's correct for the user's operating system, and adds that directory to the PYTHONPATH variable. More explicitly, if c699util exists in this location:

`~/labcode/699util/c699util/osx/c699util.py`

`~/labcode/699util/c699util/osx/_c699util.so`

then the directory `${HOME}/labcode/699util/c699util/osx` will be added to the user's PYTHONPATH, so that the `import c699util` code inside of a Python script can know to look in this directory.

As mentioned in a previous section, `_c699util.so` has some Qt4 dependencies. When `import c699util` is executed, some code at the top of c699util.py will determine if _c699util.so's dependencies are unresolved, and if so, change the dependencies to point to the correct location on the user's computer.

699util Package Release

Non-developers (i.e., users) install 699util by mounting a 699util .dmg file and running the `install699util.sh` script. The package release scheme does not use the aforementioned `~/config699/bash_config.sh` script. The .dmg file contains the following in its root directory:

- _c699util.so
- 699util.py
- Qt4 dependencies

`install699util.sh` simply copies these files into the virtual environment folder `~/py699/lib`, and then tweaks the user's .bashrc file so that `~/py699/lib` is always in the users' PYTHONPATH whenever a new terminal is opened.

Revision #4
Created 22 March 2023 17:33:58 by Nick Dobson
Updated 26 January 2026 22:13:46 by Bradley Tse