|  |  |
| --- | --- |
|  | AUSTRALIAN ASTRONOMICAL OPTICS  Facility of Science, Macquarie University.  **Version 1.0** |

Using the DRAMA2 AAO Task Template

**Contents**

1 Introduction 2

1.1 Other documentation 2

1.2 Using the acmmCreateSubsystem command 2

1.3 Basic build and run of the task. 2

2 Modifying the code. 5

2.1 Include file. 6

2.2 Main .cpp file 7

2.3 Implementation .cpp file 7

2.3.1 Polling Support 8

2.4 Fits Header support .cpp file 9

2.5 dmakefile 9

2.6 Error definition file 10

2.7 test.expected 11

2.8 Other files 11

Revisions:

11-Oct-2018: TJF, First Version

11-Jun-2021: TJF, Fix some flaws, add more implementation details.

02-May-2022: TJF, Updated for move to GitLab. Renamed from .doc to .docx.

# Introduction

This document describes how to work with the Template AAO DRAMA2 task obtained by forking <https://dev.aao.org.au/rds/templates/drama-2-template.git> (Hope to use it as a template task)

The task generated by this will easily load into many of our systems, the 2dF, KOALA, Taipan and Veloce Control tasks can all load it, initialize it and start getting FITS header information from it – it just simply needs to be added to the configuration file of the relevant control task.

But it is a template, it does nothing real. How do you turn it into your own task which does something real.

## Other documentation

First – please note that this does not replace the DRAMA2 Documentation (E.g. the [DRAMA 2 Book](https://dev.aao.org.au/rds/drama/drama/-/blob/master/drama_source/Drama2Examples/drama2.doc), in ACMM sub-system Drama2Examples, file Drama2.doc), DRAMA Web pages (<http://drama.aao.org.au>), or our growing set of Confluence pages on DRAMA 2 – (<https://confluence.aao.org.au/display/AAOSOFT/DRAMA>). Please refer to those, in particular, the DRAMA 2 book is a good introduction to DRAMA for C++ programs. But these are often general in nature, this document and the associated template code gives you a real task to start from.

## Creating the task from the template repository.

Having forked the “drama-2-template” repository, you need to clone your fork and then you must run the “RUN\_ME” script.

This script will:

1. Properly load a sub-module ([DramaTemplateDriver](https://dev.aao.org.au/rds/drama/DramaTemplateDriver)) which contains the main script.
2. Run DramaTemplateDriver/SetupTemplate which does the remainder of the work
3. It will prompt you for the name of your task. This should be a relatively short name that your DRAMA task should be known as. Keep this short – it is used as the prefix for file names and class names.
4. Any file name in the template is renamed appropriately to match your sub-system.
5. In each of the files, the template string is replaced by your sub-system name, except that case is maintained appropriately.
6. A DRAMA Error facility code will be allocated and inserted into you task. This works correctly if you have write access to the [DramaErrorCodes](https://dev.aao.org.au/rds/drama/DramaErrorCodes) repository. If not, you will need to talk to an AAO MQ staff member to get a unique error code for your task
7. Delete the DramaTemplateDriver sub-module and the RUN\_ME script.
8. Add all the changes into Git and commit them.

In in the example above, you will end up with a sub-system named “MyTask”, the task name will be “MYTASK”, the files will be named things like mytask.cpp and the namespace used will be “mytask”. Whilst everything but the sub-system name can be changed, it is best to think through the name properly before starting.

## Basic build and run of the task.

Having created repository, clone it if you don’t already have it local.

Then, presuming you have drama enabled (dramastart has been run), you can build and test your program with

cd MyTask # MyTask = repository name

dmkmf

make

The output should be something like: (don’t bother reading it all)

>> dmkmf

Build target is "linux\_x86"

Running imake to create DRAMA Makefile.

>> make

Running messgen on mytask\_err.msg: [OK]

Compiling mytask.cpp: [WARNING]

mytask.cpp: In member function 'void mytask::Task::InitialiseAction(drama::thread::TAction\*, const drama::sds::Id&)':

mytask.cpp:262:10: warning: variable 'resetHard' set but not used [-Wunused-but-set-variable]

bool resetHard = false;

^

Creating C++ dependency Makefile mytask.d from mytask.cpp

Compiling mytaskfits.cpp: [OK]

Creating C++ dependency Makefile mytaskfits.d from mytaskfits.cpp

Compiling mytaskversion.cpp to mytask\_r1\_1.o

gcc -c -pipe -fpic `$DUL\_LIB/dul\_cc` -I$DRAMA2\_DIR -I$DRTF\_DIR -Dunix -DDPOSIX\_1 -D\_GNU\_SOURCE -DDHAS\_XINERAMA -D\_FILE\_OFFSET\_BITS=64 -m32 '-DMYTASK\_VER="r1\_1"' -DMYTASK\_DATE=\"`date +%d-%b-%Y`\" mytaskversion.cpp

mv mytaskversion.o mytask\_r1\_1.o

rm -f libmytask.a

ar cq libmytask.a mytask.o mytaskfits.o mytask\_r1\_1.o

ranlib libmytask.a

Compiling mytaskmain.cpp: [OK]

Creating C++ dependency Makefile mytaskmain.d from mytaskmain.cpp

rm -f mytask

Linking mytask: [OK]

rm -f test.out test.out.tmp

IMP\_SCRATCH=`pwd` ; export IMP\_SCRATCH ; if [ "`$DITS\_DEV/tasks`" != "" ]; then ($IMP\_DEV/cleanup >/dev/null 2>&1 ; sleep 1) ; fi

IMP\_SCRATCH=`pwd` ; export IMP\_SCRATCH ; IMP\_MASTER\_NO\_NET=1 ; export IMP\_MASTER\_NO\_NET ; $IMP\_DEV/master 2>/dev/null 1>/dev/null & IMP\_SCRATCH=`pwd` ; export IMP\_SCRATCH ; $IMP\_DEV/cleanup >/dev/null 2>&1 ; sleep 2

/bin/sh: line 1: 24797 Killed $IMP\_DEV/master 2> /dev/null > /dev/null

IMP\_SCRATCH=`pwd` ; export IMP\_SCRATCH ; $DITS\_DEV/ditsloadcmd -mayload ./mytask

IMP\_SCRATCH=`pwd` ; export IMP\_SCRATCH ; $DITS\_DEV/ditscmd -n DITSCMD MYTASK SIMULATE\_LEVEL FULL > test.out.tmp

IMP\_SCRATCH=`pwd` ; export IMP\_SCRATCH ; $DITS\_DEV/ditscmd -n DITSCMD MYTASK INITIALISE | grep -v "Opened log file" >> test.out.tmp

IMP\_SCRATCH=`pwd` ; export IMP\_SCRATCH ; $DITS\_DEV/ditscmd -n DITSCMD MYTASK SIMULATE\_LEVEL REPORT >> test.out.tmp

IMP\_SCRATCH=`pwd` ; export IMP\_SCRATCH ; $DITS\_DEV/ditscmd -n DITSCMD -x MYTASK EXIT >> test.out.tmp

IMP\_SCRATCH=`pwd` ; export IMP\_SCRATCH ; if [ "`$DITS\_DEV/tasks`" != "" ]; then ($IMP\_DEV/cleanup >/dev/null 2>&1 ; sleep 1) ; fi

diff test.out.tmp test.expected

mv test.out.tmp test.out

mytask test complete.

The last set of lines in the above log are all around regression testing. The program is started, set to simulation mode and initialized. The “REPORT” action is sent to it and then it is shutdown. After all of that, the results are compared to those in the file “test.expected”. If they didn’t match, the build would fail. Since they match, all is ok.

You can even release it with “make release”, enable the sub-system with “make enable”. This will make it available to others using the same DRAMA build.

But at this stage, it is more interesting to run the task by hand. For example

./mytask &

ditscmd MYTASK SIMULATE\_LEVEL FULL

ditscmd MYTASK INITIALISE

ditscmd MYTASK REPORT

Note that when you to the INITIALISE, one of the things it does is open a log file. It will output the name of the log file. The location is set by your DRAMA\_LOGDIR environment variable value.

At this point (before exiting the task) you may be interested in looking which actions and parameters the task already has

A DRAMA control message can be used to list all the actions supported:

>> ditscmd -c MYTASK LISTACTALL

DITSCMD\_b9be:MYTASK:All supported actions of this task

DITSCMD\_b9be:MYTASK:REPORT (Report on task status)

DITSCMD\_b9be:MYTASK:EXIT (Shutdown the task)

DITSCMD\_b9be:MYTASK:RESET (Reset task)

DITSCMD\_b9be:MYTASK:INITIALISE (Initialise task)

DITSCMD\_b9be:MYTASK:UPDATE\_DRT

DITSCMD\_b9be:MYTASK:UMONITOR (DRAMA2 GIT default action implementation)

DITSCMD\_b9be:MYTASK:UPDATE\_NBD (DRAMA2 GIT default action implementation)

DITSCMD\_b9be:MYTASK:STATUS (DRAMA2 GIT default action implementation)

DITSCMD\_b9be:MYTASK:POLL\_PARAMETER (DRAMA2 GIT default action implementation)

DITSCMD\_b9be:MYTASK:DUMP\_LOG (DRAMA2 GIT default action implementation)

DITSCMD\_b9be:MYTASK:CTRLC (DRAMA2 GIT default action implementation)

DITSCMD\_b9be:MYTASK:SIMULATE\_LEVEL (DRAMA2 GIT default SIMULATE\_LEVEL action implementation)

DITSCMD\_b9be:MYTASK:POLL (DRAMA2 GIT default POLL action implementation)

DITSCMD\_b9be:MYTASK:LOG\_LEVEL (Change task logging level - DRAMA 2 action)

So there are quite a lot already. See the GIT documentation for details, but your real interest is in INITIALISE, RESET, EXIT, REPORT and POLL. The others can be ignored. In the current AAO DRAMA style, INITIALISE and RESET are almost identical, and, in the template, these are implemented with the same code.

To see all the parameters, you might fetch the special parameter \_NAMES\_ or \_ALL\_. The later brings back the values as well. E.g.

>> ditscmd -gv MYTASK \_ALL\_

DITSCMD\_6fe8:SdpStructure Struct

DITSCMD\_6fe8: LOG\_LEVEL Char [16] "ERRORS,INST,MSG"

DITSCMD\_6fe8: GITLOG\_FILENAME Char [48] "/home/tjf/scratch/logs/MYTASK-2018-10-10.02.log"

DITSCMD\_6fe8: SIMULATE\_LEVEL Char [5] "FULL"

DITSCMD\_6fe8: TIME\_BASE Float 1

DITSCMD\_6fe8: ENQ\_DEV\_TYPE Char [4] "IDT"

DITSCMD\_6fe8: ENQ\_DEV\_DESCR Char [21] "MYTASK template task"

DITSCMD\_6fe8: ENQ\_VER\_NUM Char [5] "r1\_1"

DITSCMD\_6fe8: ENQ\_VER\_DATE Char [12] "11-Oct-2018"

DITSCMD\_6fe8: ENQ\_DEV\_NUMITEM Int 0

DITSCMD\_6fe8: INITIALISED Int 1

DITSCMD\_6fe8: POLL\_PARAMETER Float 2

DITSCMD\_6fe8: FITS\_SETUP Struct

DITSCMD\_6fe8: TASKNAME Char [7] "MYTASK"

DITSCMD\_6fe8: RUN\_ID Int 0

DITSCMD\_6fe8: FITS\_KEYWORDS Struct

DITSCMD\_6fe8: FITS\_START Struct

DITSCMD\_6fe8: TASKNAME Char [7] "MYTASK"

DITSCMD\_6fe8: RUN\_ID Int 0

DITSCMD\_6fe8: FITS\_KEYWORDS Struct

DITSCMD\_6fe8: FITS\_END Struct

DITSCMD\_6fe8: TASKNAME Char [7] "MYTASK"

DITSCMD\_6fe8: RUN\_ID Int 0

DITSCMD\_6fe8: FITS\_KEYWORDS Struct

DITSCMD\_6fe8: DRT\_FINISHED Struct

DITSCMD\_6fe8: TASKNAME Char [7] "MYTASK"

DITSCMD\_6fe8: RUN\_ID Int 0

DITSCMD\_6fe8: DRT\_STRUCTURES Char [31] "FITS\_SETUP,FITS\_START,FITS\_END"

DITSCMD\_6fe8: STRING\_PARAM Char [9] "A String"

DITSCMD\_6fe8: INT\_PARAM Int 23

DITSCMD\_6fe8: SHORT\_PARAM Short 0

DITSCMD\_6fe8: DOUBLE\_PARAM Double 0

DITSCMD\_6fe8: STRUCT\_PARAM Struct

DITSCMD\_6fe8: RA Double 0

DITSCMD\_6fe8: DEC Double 0

DITSCMD\_6fe8: AZIMUTH Double 0

DITSCMD\_6fe8: ELEVATION Double 0

DITSCMD\_6fe8: HA Double 0

DITSCMD\_6fe8: ZD Double 0

DITSCMD\_6fe8: AIRMASS Double 0

DITSCMD\_6fe8: VIGNETTED Ushort 1

DITSCMD\_6fe8: RA\_STR Char [1] ""

DITSCMD\_6fe8: HA\_STR Char [1] ""

DITSCMD\_6fe8: DEC\_STR Char [1] ""

DITSCMD\_6fe8: TEL\_STATE Char [8] "INITIAL"

In the above, the parameters STRING\_PARAM, INT\_PARAM, SHORT\_PARAM, DOUBLE\_PARAM and STRUCT\_PARAM are examples. You should remove those. The others are all defined by GIT. You can see that some, e.g. ENQ\_DEV\_DESCR, have values you will want to change. We will get to this later.

# Modifying the code.

So whilst this example is a great bit of DRAMA2 code 😉, and meets the AAO standards, it probably does little else of interest to you.

Let us work through the files and where you might want to modify them.

## Include file.

In the above example, this is named “mytask.hh” It declares the C++ namespace “mytask”, and all C++ names declared in this source will be found in that namespace.

It then declares two external variables (within the namespace)

/\*\* mytask version number, from GIT\_REPO\_RELEASE macro in Makefile \*/

extern const char \* MyTaskVersion;

/\*\* mytask build date, from compiler \*/

extern const char \* MyTaskDate;

These are used to store a Git repository-based Version Number and Build date, which are later put in the ENQ\_VER\_NUM and ENQ\_VER\_DATE parameters. This all happens automatically (given the supplied dmakefile and mytaskversion.cpp file), you don’t need to change them, but do leave them be. See section 2.5.1, below, on updating this

Next the class ”MyTaskFitsClass” is declared. This is a sub-class of drtf::FitsInterface and is used to allow your task to provide FITS header information. See document ASD82 for details on this. This class is implemented by mytaskfits.cpp.

Then you have the definition of the “Task” class. This is the object which implements your DRAMA task. This is a sub-class of “drama::git::Task”, which means it is a DRAMA GIT task. See the DRAMA 2 Book for more details on this class.

The class starts declaring its member variables. The first three are:

/\* Access to the parameter system \*/

drama::ParSys \_parSys;

/\* FITS header interface \*/

MyTaskFitsClass \_fitsInterface;

/\* Are we in simulation mode.? \*/

bool \_isSimulating = false;

The \_parSys variable provides access to the task parameters and \_fitsInterface implements the FITS header interface. And the \_isSimulating variable is used to remember if the task is running in simulation mode (AAO standards require that tasks support simulation).

Two patterns should be noted here as they are encouraged. Member variables should be clearly differentiated from non-member variables, here by prefixing with an underscore. Some people have used a suffix underscore. Second, simple variables like \_isSimulated should be initialized in the class declaration – to ensure they are done.

Then there are a set of example DRAMA task parameter declarations:

drama::Parameter<std::string> \_mytaskStringParam;

drama::Parameter<INT32> \_mytaskIntParam;

drama::Parameter<short> \_mytaskShortParam;

drama::Parameter<double> \_mytaskDoubleParam;

drama::Parameter<drama::sds::Id> \_mytaskStructParam;

You should remove all of these and implement your own as required

Then there is the declaration of CreateMyStrctureParam(), an example of a method which creates a structured parameter. These must be static members since they are invoked from the constructor before anything else is constructed.

Then you will see the declaration of the various action implementation methods. InitialiseAction() and ExitAction() will implement threaded actions, whilst ReportAction() is not threaded. See the DRAMA 2 book for details on how these are different.

void InitialiseAction(drama::thread::TAction \*threadAct,

const drama::sds::Id &obeyArg);

void ExitAction(drama::thread::TAction \*threadAct,

const drama::sds::Id &obeyArg);

drama::Request ReportAction(drama::MessageHandler \*);

Finally, there is the task constructor declaration.

## Main .cpp file

This is the file that implements main(). In the above example, this is named “mytaskmain.cpp” and is very simple. The main() function just constructs the task:

drama::CreateRunDramaTask<mytask::Task>(

"MYTASK", /\* Task name \*/

"MYTASK", /\* Log sys name \*/

"MYTASK template task");

The use of “CreateRunDramaTask” ensures that any exceptions are properly reported. It creates an object of the type mytask::Task, which must be a sub-class of drama::Task. It then invokes that object’s RunDrama() method (provided by DRAMA), which runs the task.

If you want to modify the task name, then do so here. You would not normally have “TASK” in the task name, since we already know it is a task. But it does make sense in this example.

The final item to this constructor is that value that goes in the ENQ\_DEV\_DESCR parameter, so you should change that.

Otherwise, the normal changes here are often to add command argument handling of some form, e.g. you might want it to be possible specify the task name as an argument.

## Implementation .cpp file

This is main implementation file, in this example named mytask.cpp. It provides the constructor implementation and the action implementations. You may want to break it into multiple files as your task grows.

My suggestion is to work through this yourself, the comments explain what is going on and what you might like to change.

Do note how logging is done (In InitiaiseAction()) and how locks are managed (again, in InitiaiseAction()). In particular, note that the DRAMA lock is taken by constructing an drama::thread::AccessDrama object rather the more general drama::task::guardType. AccessDrama ensures that the DRAMA action context is set correctly to the threaded action you are running, ensuring that messages to the user are sent correctly to the task which sent the obey message. Think carefully when reading/writing variables accessed by other threaded actions.

Note the availability of the “CheckInitialised” method. This should be invoked as the first thing in any action which requires that the task be initialized (e.g. the normal actions of the task other then INITIALISE, RESET and REPORT).

### Polling Support

The example does NOT show any support for polling of hardware status, through it does inherit a default implementation of POLL from the GIT implementation.

That GIT implementation is from drama::git::Task and provides various ways to quickly implement common requirements. You can implement the following methods to override the behavior of the default POLL action as required

|  |  |
| --- | --- |
| **Method** | **Description** |
| PollObeyOverride() | This method allows the sub-class to override specific poll action operations. The default implementation does nothing and returns false.  Return true if the message has been handled and false to indicate the default handling should happen. This allows the user code to handle only some messages, and leave the rest to the defaults.  See the drama::git::Task documentation for more details. |
| PollKick() | This method allows the sub-class to override specific poll kick operations. The default implementation does nothing but message the user and returns true, which would cause the POLL action to complete. See the drama::git::Task documentation for more details. |
| PollRescheduleEvent() | This method is invoked if the POLL action receives a reschedule event, but only if PollObeyOverride() returned false for this event. |
| PollSignalEvent() | This method is invoked if the POLL action receives a signal event, but only if PollObeyOverride() returned false for this event. |
| PollRescheduleRequest() | Used to change the default rescheduling of the POLL action. |

A sub-class can read/write the class member \_pollPar to access or change the rescheduling interval (floating point seconds). This is what determines how often PollRescheduleEvent() is invoked.

See the documentation for drama::git::Task for more details.

Alternatively, you could just re-implement POLL entirely if you prefer by adding a new action handler for it.

## Fits Header support .cpp file

This implements the task’s FITS headers, and in this example is named mytaskfits.cpp. It does this by implementing the class ”MyTaskFitsClass”, which was declared in the main header.

See document ASD82 for details on what is being done here. The class is a sub-class of the DRTF FitsInterface class. Other than linking keywords to parameters or providing the required information there isn’t really much for you to do. It is a bit more difficult if you are implementing say a multiple arm spectrograph. See the DRTF documentation for more detail on that.

It is strongly recommended that you provide changing FITS header information via parameters which are linked to the keyword in this module. That means that other parts of you task can just update the parameter. But this does fall apart if keywords are not always present or have complex values or strings. In those cases, you can add them in the Start() and End() methods.

Remember that all keyword names must be 8 characters or less long and must be unique in the FITS file, so you need to check against standard for the system you will be part of. *NOTE that the default names for the task version number and date are probably longer the 8 characters after applying the template to the original code, you will need to modify them appropriately (mytaskfits.cpp, ::Setup() method).*

## dmakefile

This file is processed by the “dmkmf” command to create a Makefile. You will need to modify it to add extra files to build, add extra include directories or libraries to link against, and to expand your tests.

It presumes you are building a C++11 file. And if you try to build the sub-system on a DRAMA build that does not support C++11, the build will fail.

At the beginning of the non-comment part of the code, you will see a section bracketed by #BeginConfig and #EndConfig. The main thing you might want to change here is the INCLUDES specification. The example is

INCLUDES = DramaIncl IDir(DRAMA2\_DIR) IDir(DRTF\_DIR)

The first value here, DramaIncl, says to pick up all the standard DRAMA include files. But since DRAMA2 was added later, that is not included, and the IDir(DRAMA2\_DIR) specification says to pick up DRAMA2. You can use the IDir() macro to grab include files from any sub-system released using DRAMA. In this case, the DRTF library is also being used. You can also add normal “-I” specifications here, but be careful that they may make your program less portable. Many commonly used libraries such as CFITSIO and SLA have their own portable include specification macros. (Look at some of the code which uses those libraries).

The next bit of interest, after #EndConfig, is the LIBS specification. This specifies the libraries to link against. Order is important – things your library wants to link against must be in libraries listed later. The example is

LIBS=LinkLib(mytask) LinkLibDir(DRTF\_LIB,drtf) LinkLibDir(DRAMA2\_LIB,drama2)

The LinkLib macro is how you pick up a library in your current directory. The LinkLibDir() macro is how you pick up other libraries released by DRAMA, DRTF and DRAMA2. The Core DRAMA and system libraries will be added automatically. You can add libraries with the -L specification, but against you risk making your program less portable. Again, many commonly used libraries such as CFITSIO and SLA have their own portable include specification macros.

The OBJECTS\_BASE specification is where you list the C++ (or C) source which makes your program. In the example, this is

OBJECTS\_BASE=mytask mytaskfits

If, for example, I found my initialise action was getting complex and moved it to another file, I would list that file here.

The final thing of interest is the test specifications. In this example, look for this line

test test.out : Lib(mytask) mytask test.expected

The first target, test, is used to ensure “make test” runs the following test. That code generates “test.out.tmp”, which is compared to test.expected as part of the sequence. If they differ, make fails. Otherwise “test.out.tmp” is cpied to “test.out” and a normal “make” command will not run the testing again unless the source has changed. So as the output of your program changes, this part of the build will fail and you will need to manually copy “test.out.tmp” to “test.expected” to continue. Of course, you would normally only do that after having confirmed the output is correct.

If you want to build and release without testing (sometimes, but rarely needed), run “dmkmf” with the “-noautotest” option.

Note that the testing sequence consists mainly of loading the task and then using “ditscmd” to send a number of actions. As you develop your task, you will probably want to send more here to test more features.

### Release number – GIT\_REPO\_RELEASE and RELEASE macros.

The release number of a DRAMA sub-system locates where the files are written to when you type “make release”. See DRAMA documentation for more information about, but the basic idea is that every new version of your task should be released in a separately in the DRAMA release directory - you never overwrite working code! The “dramastart” command can specify version files which selects sets of working versions. How all this works is for elsewhere, but here we need to talk through how you get a new version number.

In the template dmakefile, just after the “#BeginConfig” line, you will see

XCOMM If your makefile fails at the following none-comment line …

XCOMM Add "#define SetGitRepoRelease() …

XCOMM If instead you have the line starting GIT\_REPO\_RELEASE …

SetGitRepoRelease() /\*...\*/

RELEASE=$(GIT\_REPO\_RELEASE)

First note that in a “dmakefile”, the “XCOMM” prefix of a line ensures the line is passed through to the “Makefile” as a comment. So those three lines all get passed through to the Makefile.

If everything works as expected, dmkmf will replace “SetGitRepoRelease()” by:

GIT\_REPO\_RELEASE:=$(shell env -u GIT\_DIR git describe –dirty)

And that value of this is then used for the RELEASE macro value. The use of two macros here is largely historical (RELEASE has always been used in DRAMA tasks, GIT\_REPO\_RELEASE is new after AAO moved to GitLab, but is not as portable)

Note that in many cases, you will simply want to remove those XCOMM lines, just leaving from “SetGitRepoRelease()” but they do help with fault finding.

#### Explaining the release number

As mentioned above – the release number comes from the “git describe --dirty” command. This requires that you have set at least one “annotated tag”. The exact value you get depends on your most recent tag, post tag commits and if your repository is dirty. The format is as followed:

<lasttag>[-<num\_commits>-<last\_commit>][-dirty]

Here:

* lasttag => is the most recent “annotated” tag name
* num\_commits => is the number of commits since that tag
* last\_commit => is the smallest most unique hash of the last commit
* “-dirty” => will be added if your repository is dirty. E.g. files have been modified or added with “git add”.

This is a real example:

b2.0.1-4-g6a374ed

Here:

* “**b2.0.1**” is the most recent “annotated” tag name (everything before the first dash)
* There have been 4 commits since the tag was created.
* “**g6a374ed**” => is the smallest most unique hash of the last commit.
* The repository is not dirty – so you can trust the commit has to give you the actual code being used.

***You MUST NEVER release outside your own local testing, software from a “dirty” repository, as it cannot be traced to source.***

##### Not yet updated AAO Sub-systems

If you find yourself working with older AAO sub-systems, you may find RELEASE has not yet been set to $(GIT\_REPO\_RELEASE), but remains set to the value $(ACMM\_RELEASE). Please add the “SetGitRepoRelease()” update the value for RELEASE.

#### Generating a tag

Tagging is simple but does have some complications. By default, a simple tag is generated, and is NOT annotated. These won’t be available to “git describe”. Additionally, you must make an extra step to push tags to a remote repository.

To create the above tag (b2.0.1) and push it to the repository, the following commands would have been executed (after a commit of all the required changes)

git tag b2.0.1 -m “annotation of the tag”

git push origin b2.0.1

#### GIT\_REPO\_RELEASE failures

##### SetGitRepoRelease() macro not available.

If your local DRAMA environment is not defining the “SetGitRepoRelease()” macro, then the macro won’t be translated when the Makefile is generated. A typical error when you run make would be

Makefile:6: \*\*\* missing separator. Stop.

When you then look at the Makefile, line 6, you would see it still contains “SetGitRepoRelease()”.

This happens if you don’t have a recent version of DRAMA (DRAMA release number 2.0 onwards). You can resolve this by either updatingRAMA, or just adding the following lines to your $DRAMA\_LOCAL/drama\_local.cf file:

#ifndef SetGitRepoRelease

#define SetGitRepoRelease() GIT\_REPO\_RELEASE:=$(shell env -u GIT\_DIR git describe --dirty)

#endif

##### Repository not tagged.

If you notice this message coming out during your build

"fatal: No names found, cannot describe anything."

Or you notes that your release number is not being defined, then it is likely your repository does not have any annotated tags. Please add them as per section 2.5.1.2, above.

## Error definition file

In this example, this is the, mytask\_err.msg. This is used to generate DRAMA compatible error codes. My non-comment part of this file is

.TITLE MYTASK\_ERR template DRAMA Task

.FACILITY MYTASK,1100/PREFIX=MYTASK\_\_

.SEVERITY ERROR

.BASE 2

MALLOCERR "Malloc error"

NOTINIT "The task has not been initialised"

.END

The contents of this file is defined by the DRAMA Message code system, see <http://drama.aao.org.au/doc/ps/mess_6.ps>.

As a result of the processing of this file when you run make, the files mytask\_err.h and mytask\_err\_msgt.h are generated. The former has the status code macros, in this case

/\* Facility number \*/

#define MYTASK\_\_FACNUM 1100

/\* OK Status symbol \*/

#define MYTASK\_\_OK 0

/\* Malloc error \*/

#define MYTASK\_\_MALLOCERR 0xC4C8012

/\* The task has not been initialised \*/

#define MYTASK\_\_NOTINIT 0xC4C801A

You can see that it has defined two codes, MYTASK\_\_MALLOCERR and MYTASK\_\_NOTINIT based on the MALLOCERR and NOTINIT specifications in the .msg file. You would normally use codes like this as your DRAMA status when you throw exceptions. You can add as many as you like

Note – don’t bother using the MYTASK\_\_OK symbol. We only ever use STATUS\_OK in our code now days.

The later provides C code which associated a string with each code, generating the structure used with MessPutFacility() in your task constructor.

WARNING:If you copy such a file from another sub-system, say if you are cloning another task to start yours, ensure the .TITLE and .FACILITY lines are the ones you got in the template. Do NOT reuse the facility number (1100 in this case) from a different sub-system. They are meant to be unique to the sub-system.

## test.expected

This is the expected output of your test run. If you change the output of your program, then the “make” will fail. You will want to check the contents of test.out.tmp are correct and copy it to test.expected.

## Other files

The version specification file, in this case, mytaskversion.cpp, is used to determine the git repository based version number and the build date. These are both set from the dmakefile when make is run.

The README.txt file is meant to introduce your task. Please provide a few paragraphs of description and any notes about anything special about your program. References to relevant documentation are also good. (If you have documents say describing how to use the hardware your task is to control, drop this in the ACMM sub-system).

And then there is the ChangeLog file. Summarize changes here and refer to any JIRA tickets you were dealing with.

And there is this file. Drama2\_TemplateTask.doc. Feel free to delete it when you are done, or (better) use it to start your overview documentation for your task (which should include action,s parameters and FITS keyword descriptions)