AUSTRALIAN ASTRONOMICAL OPTICS

Macquarie University

MACQUARIE University
SYDNEY·AUSTRALIA

DRAMA

revision 12

Table of Contents

Exported on Jan 04, 2024

Top-level for DRAMA related pages. All DRAMA pages should have this page as their top-level. Add the tag "drama" to make a page appear in this table.

Title	Creator	Modified
DRAMA 2 Reporting/Logging Notes	Tony Farrell	Aug 30, 2023
Machine Configuration for DRAMA systems.	Tony Farrell	Aug 30, 2023
DRAMA - releasing subsystems (AAT style)	Tony Farrell	Dec 02, 2022
DRAMA and AAT/UKST Software Instrumentation Documentation	Tony Farrell	Dec 02, 2022
DRAMA Modifications	Tony Farrell	Aug 18, 2021
Ensuring translations of error codes.	Tony Farrell	Feb 06, 2020
DRAMA2 C++ ACMM Template	Tony Farrell	Oct 11, 2018
DRAMA_overview	Tony Farrell	Sep 03, 2018
DRAMA refresher course topics.	Tony Farrell	May 21, 2018
dmkmf/dmakefile tricks and treats	Tony Farrell	May 18, 2018
The SDS Compiler - creating/using SDS structures matching C structures - with DRAMA2	Tony Farrell	May 15, 2018
DRAMA 2 - ensuring ERS, "gitarg" classes, and ERS from old libraries work from threads.	Tony Farrell	Apr 11, 2018
DRAMA - Control messages - how to work out what a task is doing!	Tony Farrell	Apr 05, 2018
DRAMA 2 - integrating C++ streams into DRAMA messages.	Tony Farrell	Jan 22, 2018
DRAMA 2 programs and exceptions.	Tony Farrell	Sep 28, 2017

1 DRAMA - Control messages - how to work out what a task is doing!

DRAMA's control messages are an additional class of messages of the same form as Obey/Kick/Set/Get/Monitor. The difference that these messages are interpreted by the DRAMA Fixed part rather then the part the author writes. So they are available in all DRAMA tasks.

1.1 Sending Control Messages

Control messages are just another message and all interfaces for sending DRAMA messages have some way of sending control messages. Through in some cases, including the basic "Dits" interface, you have to use a generic message sending routine rather then a particular routine, that is, there is no `DitsControl()` in the same fashion as `DitsObey()` and `DitsKick()`, instead you must invoke the underlying `DitsInitiateMessage()`. But most of the time, you will likely find the "`ditscmd`" sufficient. You send a Control message using its "-c" option. E.g.

> *>> ditscmd -c TICKER DEFAULT*
> *DITSCMD_2d4e:/net/fileserver/iscsi/home/tjf/xxx_stuff/drama/DramaDits*

The "DEFAULT" control message returns a task's working directory. You can actually change this by specify an argument the message with the name of the new working directory. (This change can be prevented by the task, if needed - see the `DitsDefault()` routine).

1.1.1 Getting a list of control messages.

If you specify an invalid control message name or "HELP", then the list of available control messages are returned. E.g.

> *>> ditscmd -c TICKER HELP*
> *DITSCMD_2faf:TICKER:Task TICKER received invalid control message HELP*
> *DITSCMD_2faf:TICKER:Valid control messages are*
> *DITSCMD_2faf:TICKER: DEFAULT MESSAGE DEBUG DUMPPATHS DUMPTRANSIDS*
> *DITSCMD_2faf:TICKER: DUMPACTACTIVE DUMPACTALL DUMPMON VERSIONS*
> *DITSCMD_2faf:TICKER: LOGNOTE LOGFLUSH LOGINFO SDSLEAKCHK HELP*
> *DITSCMD_2faf:TICKER: PRINTENV*

1.2 The control messages

Message	Description
DEFAULT	If no argument is specified, returns the current working directory of the task. If an argument is specified, attempt to make it the new working directory for the task. This later behaviour can be overridden using the `DitsDefault()` routine.
MESSAGE	Requires one argument which is an integer value (with hexadecimal formats accepted if prefixed by 0x). The integer value is translated as a DRAMA error code and the resultant string is returned. This message could be used to translate error codes that a user interface may not know about, but which are known by the task.
DEBUG	Set the DITS Internal debugging flag. This determines how much logging DITS will output, which goes into the log file if there is one, but otherwise goes to standard error. See `DitsDebug()` for more details.
DUMPPATHS	Dump the list of paths (connections to other task) this task has.
DUMPTRANSIDS	Dump the list of transactions to other tasks which are outstanding. E.g. when a message has been sent to another task but the final reply has not yet been received.
DUMPACTACTIVE	Dump the list of actions which are currently running.
DUMPACTALL	Dump the full list of actions available.
DUMPMON	Dump the list of parameters which are being monitored and the names of the tasks which are monitoring them.
VERSIONS	Output build version information for DITS, IMP and DRAMA itself.
LOGNOTE	The argument is a string which is written to the task's log file, if it has one.
SDSLEAKCHK	Outputs information on SDS Id's being used. By running this occasionally you can work out if a task is leaking SDS ID's, which will eventually cause the task to slow down.
HELP	Print the list of control messages.
PRINTENV	Print a list of all of the tasks environment variables
LISTACTALL	List the actions of a task. Similar to DUMPACTALL, but only action names, rather then all details on each action, are output.
LISTACTACTIVE	List the actions of a task. Similar to DUMPACTACTIVE, but only action names, rather then all details on each active action, are output.

1.3 Examples

Translate an error code.

> *>> ditscmd -c TICKER MESSAGE 0xC248182*
> *DITSCMD_3509:%DRAMA2-E-SIGKICK, Signal event received when expecting a kick message*

It so happens that the above was the DRAMA2 example of the "ticker" program. It new how to translate a DRAMA2 error code.

Dump the list of active paths .

```
>> ditscmd -c TICKER DUMPPATHS
DITSCMD_3538:TICKER:Dump of paths held by task TICKER
DITSCMD_3538:TICKER:Path 0x9444f08 to task DITSCMD_3538@aaolxp.aao.gov.au (3538@1b0a500a)
DITSCMD_3538:TICKER: Status PSTAT_OK, Imp connection number 3, old con num -1
DITSCMD_3538:TICKER: local, using SDS, Flags = 0x100
DITSCMD_3538:TICKER: Path was set up by other task
DITSCMD_3538:TICKER: - hence initial buffer spec not known
DITSCMD_3538:TICKER: Total buffer size is 8544, Free Bytes 7927, Unread messages 1
DITSCMD_3538:TICKER:Path 0x9444ad8 to task DTCL@aaolxp.aao.gov.au (30f6@1b0a500a)
DITSCMD_3538:TICKER: Status PSTAT_OK, Imp connection number 2, old con num -1
DITSCMD_3538:TICKER: local, using SDS, Flags = 0x0
DITSCMD_3538:TICKER: Path was set up by other task
DITSCMD_3538:TICKER: - hence initial buffer spec not known
DITSCMD_3538:TICKER: Total buffer size is 8544, Free Bytes 8544, Unread messages 0
DITSCMD_3538:TICKER:Path 0x94443e0 to task TICKER@aaolxp.aao.gov.au (30f7@1b0a500a) (path To self)
DITSCMD_3538:TICKER: Status PSTAT_OK, Imp connection number 0, old con num -1
DITSCMD_3538:TICKER: local, using SDS, Flags = 0x0
DITSCMD_3538:TICKER: Path was set up by task itself
DITSCMD_3538:TICKER: Initial buffer spec 2000 1 0 0
DITSCMD_3538:TICKER: Total buffer size is 2144, Free Bytes 2144, Unread messages 0
>>
```

The above shows TICKER has three paths. The first one is to "DITSCMD_3538". You can see that this path has one unread message. This is actually the version of DITS that sent the control messages, it has an unread message since it has not yet finished reading the relevant message. There are two other paths, one to "DTCL" (which was connected at the time) and the other to the task itself. In all cases,you can see some details about how the task was send up and some buffer sizes. Buffer size information is somewhat dependent on which task set of the message.

Dump the list of actions in the ticker program:

```
>> ditscmd -c TICKER DUMPACTALL
DITSCMD_3539:TICKER:Dump of actions in task TICKER
DITSCMD_3539:TICKER: Task supports 3 different actions.
DITSCMD_3539:TICKER:Action EXIT, Index 0, is not Active
DITSCMD_3539:TICKER: Obey Routine 80f9354, Kick Routine 80f93ee, Code 155469632, Cleanup Routine 80f9fb4
DITSCMD_3539:TICKER: Action data value is 0, Transaction Count = 0
DITSCMD_3539:TICKER:Action TICK, Index 1, is not Active
DITSCMD_3539:TICKER: Obey Routine 80f9354, Kick Routine 80f93ee, Code 155470424, Cleanup Routine 80f9fb4
DITSCMD_3539:TICKER: Action data value is 0, Transaction Count = 0
DITSCMD_3539:TICKER:Action LOG_LEVEL, Index 2, is not Active
DITSCMD_3539:TICKER: Obey Routine 8108864, Kick Routine 0, Code 155469224, Cleanup Routine 0
DITSCMD_3539:TICKER: Action data value is 0, Transaction Count = 0
```

You can see there are three actions, two of which have "Kick Routines" and one of which does not. The other information (Code, Cleanup Routine) is useful if you know a bit more about the internals of DRAMA.

List the parameters being monitored

```
>> ditscmd -c TICKER DUMPMON
DITSCMD_3541:TICKER: LOG_LEVEL: DTCL
>>
```

The above shows that the TICKER task LOG_LEVEL parameter is being mounted by the "DTCL" task.

It so happens that TICKER, being a simple client, won't have any outstanding transactions, so I sent the relevant message to the DTCL task which is monitoring the LOG_LEVEL parameter in TICKER

```
>> ditscmd -c DTCL DUMPTRANSIDS
DITSCMD_3584:DTCL:Dump of transactions in task DTCL
DITSCMD_3584:DTCL:UFACE Transactions
DITSCMD_3584:DTCL:Transaction 0x965cd9c of type DITS_MSG_MONITOR to task TICKER (path 0x965cc98 -
aaolxp.aao.gov.au)
DITSCMD_3584:DTCL: Uface transaction, routine 8076b78, code 9660710
```

```
DITSCMD_3584:DTCL: Unique ID = 0, Obey ID = 0, Tag = 1003
DITSCMD_3584:DTCL: Monitor Transaction Arg is "LOG_LEVEL "
DITSCMD_3584:DTCL:There were 1 UFACE transactions
DITSCMD_3584:DTCL:There were 1 outstanding transactions
DITSCMD_3584:DTCL:Transaction id memory dump, size of tid = 76 bytes
DITSCMD_3584:DTCL:Transaction id memory buffer 0x965c900
DITSCMD_3584:DTCL: Buffer 0x965cd50, buffer end 0x965cecc
DITSCMD_3584:DTCL:NextAvail = 0x965cde8, last = 0x965cecc, total = 5, alloc = 2
DITSCMD_3584:DTCL:Buffer Contains
DITSCMD_3584:DTCL: 0x965cd50
DITSCMD_3584:DTCL: Num on free list is 1
DITSCMD_3584:DTCL:Note that deleted but locked items are NOT on the free list
>>
```

The important thing here is that there is on UFACE (generated by user interface code) transaction outstanding. This is a MONITOR message to TICKER. There is a fair bit of information about the transaction list memory management (which is a little complicated for efficiency reasons) intended only for DRAMA internals debugging.

Now I check what versions TICKER was built against.

```
>> ditscmd -c TICKER VERSIONS
DITSCMD_38fe:TICKER:DITS library version "r3_106", build date "Nov 1 2017"
DITSCMD_38fe:TICKER:DRAMA Version "1.6.1" (during DITS build)
DITSCMD_38fe:TICKER:IMP Build Ver "r1_6_3_64", Build Date "Aug 4 2017", Internal Ver "5.1"
>>
```

That would make it pretty easy to find the actual source code for the DITS and IMP libraries. The DRAMA version does not link quite as well to the source code versions.

2 DRAMA 2 programs and exceptions.

First note that there are relevant examples and explaining in the DRAMA 2 book (ACMM sub-system `Drama2Examples`, file `drama2.doc`). Section 3 of the DRAMA 2 Book gives the introduction to this, whilst the example in section 7.13 (example 7-12) happens to show the use of the futures.

2.1 Uncaught C++ Exceptions and DRAMA2 programs

Any uncaught C++ exception causes a program to terminate. We normally don't want this to happen, we want errors to be handled in sensible ways, particularly things like argument or command sequence errors, or things that can be attempted again by the user.

In DRAMA2, the `main()` function should do something like below to get the better error messages. Otherwise it will terminate with a very simple error message that doesn't normally help and a core dump. That is unless it uses `drama::CreateRunDramaTask` to run the task, which does all of this (this is the preferred approach, but not always possible)

```
int main()
{
 try
 {       DramaExampleTask task("EXAMPLE3_2");       task.RunDrama();
 }
 catch (const drama::Exception &e)
 {        std::cerr << "drama::Exception caught by main()"
 << std::endl
 << e // Outputs all the exception details.
 << std::endl;       exit (e.statusAsSysExitCode());
 }
 catch (const std::exception &e)
 {       std::cerr << "std::exception caught by main()."
 << std::endl
```

```
  << e.what()
  << std::endl;        exit(1);
  }
  catch (...)
  {          std::cerr << "Non-standard exception in main()." << std::endl;
  throw;
  }
  return 0;
}
```

If a non-threaded DRAMA action throws, DRAMA will catch that exception and the action will fail with bad status and generate an ERS error report with the exception details.

If a threaded DRAMA action throws, the same things happens, but the details of how that is implemented are a bit more complicated.

But - if you create your own threads, you must deal with it somehow. If you create a thread using, for example, `std::thread`, then you are on your own, and an uncaught exception will cause the program to terminate. What I do instead is use `std::async()` to start the thread, which returns a std::future. If the thread throws, the exception is stored in the future. The creator of the thread can do a `get()` on the `future`, and the exception is thrown at that point - so you transfer the exception to the creator of the thread. If the creator is a DRAMA thread, it is then handled correctly. See the DRAMA2 book example 7-12 for an example of doing this.

2.1.1 Other causes of exceptions crashing programs.

Another cause of an exception terminated a program is throwing an exception whilst handling an exception. That can also causes the program to terminate (or may simply confuse you about the cause or the exception depending on where the code is when it happens).

Another one is throwing an exception from a destructor - that causes a program to terminate.

3 DRAMA Modifications

This page lists recent modifications to DRAMA. Roughly from September 2017. The intention here is to report interface changes or additions, rather then bug fixes, but some of those may also be reported if widely seen.

3.1 General Changes

3.1.1 Various Sub-systems (Aug 2021) - Mac M1 support

Support building DRAMA for the ARM64 Architecture on a Mac M1 (or similar). Note this depends on the architecture your shell indications you are running in, which can be changed. E.g. to make it think it is an intel machine, you do

```
"arch -x86_64 $SHELL"
```

To make it think it is arm, you do

```
"arch -arm64 $SHELL"
```

If your apparent architecture is ARM64, then the build will default to ARM64.

If you want to build for intel, run the "arch" command above and then set

```
DRAMA_MAC_INTEL=1
```

Before you do the build.

Do note that you need any libraries you are using (E.g. Tcl/Tk) compiled for the right architecture with the right versions selected in the `$DRAMA_LOCAL/local/drama_local.cf` file (or those it includes).

Changes to `DramaConfig`, `sds` and `messgen` sub-systems.

3.1.1.1 Potential build issue

An issue has been seen when building for ARM64 on an M1. The build will fail in the GIT sub-system regression tests. It is not clear at the moment if this is M1, OS version or DRAMA specific, but feels like an OS flaw of some form. Please talk to Tony Farrell if you have such a failure. If you are NOT running the regression tests, the same problem would be seen when you later execute "`ditscmd`", the problem sees it crash early (e.g. execute it without arguments and rather then complaining, it just crashes).

3.2 DITS Changes

3.2.1 From DITS version 3.112 (April 2020)

You can now add a description to an action which is output by the `LISTACTALL` and `LISTACTACTIVE` actions. Changes to `DitsPutActions()` support this and new methods `DitsPutActDescr()`, `DitsPutThisActDescr()` and `DitsGetActDescr()`.

Add new `DitsPutAction()` routine which allow you to add a single action

3.2.2 From DITS version 3.107 (March 2018)

Add `LISTACTALL` and `LISTACTACTIVE` control messages, simple versions of `DUMPACTALL` and `DUMPACTACTIVE` which just provide the action names.

3.3 DRAMA 2 - other changes (other then new methods, see below)

New methods are listed below. This section lists other changes that may impact user code (as against simple bug and build fixes)

3.3.1 From DRAMA 2 version 1.118 (Jun 2021)

Add the `impdir` flag to `drama::FindFile()`.

3.3.2 From DRAMA 2 version 1.115 (May 2021)

The gitarg::Enum class modified to ensure possible values are output in error messages when the supplied value is invalid.

Fixed a line splitting problem in `Exception::AddErs()`.

3.3.3 From DRAMA 2 version 1.112 (Dec 2020)

TUFace constructor will now throw if you are creating this within the thread running the DRAMA main loop - catches a particular class of issues. Ticket ASI-162 - DRAMA2 TUface context items should not be created in DRAMA thread. DONE

3.3.4 From DRAMA 2 version 1.110 (Dec 2020)

drama::Exception() main constructor modified to take an extra argument which indicates the strack trace is always to be output/

3.3.5 From DRAMA 2 version 1.108 (Sep 2020)

SignalBlocker class and drama::thread::BlockSignals() method, no longer block SIGABRT. Blocking this signal stops assert() killing the program. Similarly, SIGSEGV, SIGILL - don't want to continue from those.

3.3.6 From DRAMA 2 version 1.101 (Oct 2019)

Simplify most Exception reports

3.3.7 From DRAMA 2 version 1.100 (Oct 2019)

Drop DramaTHROW_F() macro. Has been marked as depreciated for some time and I don't believe anyone was using it.

3.3.8 From DRAMA 2 version 1.97 (Jun 2019)

Support building with C++17 if available.

3.3.9 From DRAMA 2 version 1.94 (Jan 2019)

TUface constructor now takes a name for the thread function, defaulting to the old name that was used, which removes the need for a complex sequence to set this.

We can now disable the warning which appears when RunDrama() is executed in a different thread from the thread in which the task was constructed. The logger now handles this properly.

3.3.10 From DRAMA 2 version 1.94 (Jan 2019)

TUface constructor now takes a name for the thread function, defaulting to the old name that was used, which removes the need for a complex sequence to set this.

We can now disable the warning which appears when RunDrama() is executed in a different thread from the thread in which the task was constructed. The logger now handles this properly.

3.3.11 From DRAMA 2 version 1.92 (Nov 2018)

drama::Logger::SLog() now takes const std::string& rather then const char *.

3.3.12 From DRAMA 2 version 1.88 (Jul 2018)

`MonitorByType` code was not handling 64 array data correctly, which it should be able to do. Fixed. Also was allowing handling
of 64 bit unsigned as a long int on 32 bit machines, which is wrong. Fixed

3.3.13 From DRAMA 2 version 1.87 (June 2018)

`drama::parSys` parameter `Get()` methods are now const

3.3.14 From DRAMA 2 version 1.85 (April 2018)

By defining the macro `DRAMA2_LOCK_DEBUG2` in `drama/task.hh` when building DRAMA2, it is now possible to build a version that writes a lock log file name `DramaLock-<pid>.log`. This is intended to help debug deadlock situations (but performance changes may mean problems go away!!) (But does require a special build of DRAMA2)

3.3.15 From DRAMA 2 version 1.84 (April 2018)

The various MessageUser() methods and the drama::ErsReport() method, will now accept format list arguments - i.e. add a "%" to the string argument and add an extra argument which can be output to a ostream and the result of that will be put in the string. For this version of ErsReport(), the status argument will be first rather the last.

3.3.16 From DRAMA 2 version 1.82 (Mar 2018)

Added drama::ErsFlushStack() and drama::ErsAnnulStack() methods. Versions of ErsFlush() and ErsAnnul() which ensure the DRAMA lock is taken.

3.3.17 From DRAMA 2 version 1.81 (Mar 2018)

Anything that invokes ERS (including `drama::ErsReport()` and various methods in `drama::Exception` and `drama::gitarg` classes now checks to make sure you have the DRAMA lock when the call is made.

WARNING: This update could cause some user code to fail. ***Such user code is faulty!*** The change to check the user code is right. See the release method documentation or this page for details.

3.3.18 From DRAMA 2 version 1.80 (Mar 2018)

Add the `drama::IsRunningType` class, a drama::MessageEventHandler sub-class that can be used to work out if a transaction is currently running. Allows one thread to work out if an action is still outstanding in another thread.

Add `Exists()` method to `drama::ParSys` Class.

The `drama::thread::KickNotifier` class was not handling signals correctly. Resolved.

3.4 DRAMA 2 - New methods

3.4.1 From DRAMA 2 Version 1.1

3.4.2 From DRAMA 2 Version 1.119 (Aug 2021)

Added the fmt library to DRAMA2. This library implements a python style formatting approach which is far more efficient and safer than the older safe print functionality (E.g. drama:SafePrintf() and other routines/methods which do similar things). It also compatible with the C++20 std::format() library (whilst being usable with C++11).

See https://fmt.dev/latest/contents.html for the documentation on this library, but note you should include is from "`drama/fmt/..`" rather then from just "`fmt/...`" and the core.h file has been renamed to fmtcore.h. The example file `Drama2/examples/msgoutfmt.cpp` shows how to use it in various cases.

3.4.3 From DRAMA 2 Version 1.117 (Jun 2021)

Added overloads to `gitarg::Int` and `gitarg::Real` classes (Constructors and Get methods) which allow the ranges to be specified when invoking rather then only at the template instantiation stage.

3.4.4 From DRAMA 2 Version 1.116 (Jun 2021)

Added drama::FindFile(), a simple interface to DulFindFile(). Recommend method for finding installed files via search paths.

3.4.5 From DRAMA 2 Version 1.112 (Dec 2020)

Add drama::ScopeGuard, as simple scope guarding class implementation, and drama::ScopeGuardAtomic which does a similar thing for classes which are instantiations of std::atomic. See examples in scopeguard_test.cpp

Add Logger::RegisterThreadNoDramaCtx() ASI-161 - DRAMA2 Logger::RegisterThread should support threads with no DRAMA context. DONE

3.4.6 From DRAMA 2 Version 1.110 (Dec 2020)

New overloads of most drama::gitarg class constructors which take a drama::thread::TAction argument as the first argument. These overloads will use that argument to lock DRAMA during the call and hence avoid problems with the DRAMA lock not being taken when ERS calls are made (see version 1.81 above).

3.4.7 From DRAMA 2 Version 1.108 (Sep 2020)

Add abstract class ThreadDispatcher and the derived class ObeyThreadDispatcher. Used to allow messages to be sent in threads, wrapping a common design pattern in DRAMA 2 control tasks. Note - other derived classes to be provided!!

3.4.8 From DRAMA 2 version 1.104 (April 2020)

Add `GetActionName()` method to `TAction` class

All methods which add actions to tasks now take an extra (optional) parameter which is a string to use as the action description. See 3.112. You can define the macro DRAMA2_ADD_ACT_NO_DESCR_EXPOSE to remove the default value, which exposes all the cases where you may want to add these.

3.4.9 From DRAMA 2 version 1.94

Replaced drama::Task::GetTaskThreadId() by GetTaskCreThreadId() and GetTaskRunThreadId().

3.4.10 From DRAMA 2 version 1.91

Add new drama::thread::Monitor constructor which takes a vector of strings. (As you can initialize a vector from an initialize list, but if you already have a vector, you can't initialize it with a vector.)

3.4.11 From DRAMA 2 version 1.90 (Oct 2018)

Add new methods to `drama::Task`, `AddMth()` and `AddMthThd()`, These can be used to add action implementations via methods of a sub-class of drama::Task. This will become the default way of implementing new actions in the DRAMA2 template task.

It should be noted that many tasks developed before this date have their own approach to this, for example, in task I (Tony) have written, I have Add() (with action name first rather then second) and AddTA() methods which do a similar thing. Such tasks should be modified to use the DRAMA2 methods, but this is not an urgent problem.

3.4.12 From DRAMA 2 version 1.86 (May 2018)

drama::git::Path class now has a RemoteNetStart() method, which can be used to start DRAMA networking on a remote node using, for example, DramaUnixStart. The default implementation does nothing. See the `TaipanControlTask` or `VeloceControlTask` source code for examples of how this is used.

3.4.13 From DRAMA 2 version 1.80 (Mar 2018)

Add the `drama::IsRunningType` class, a drama::MessageEventHandler sub-class that can be used to work out if a transaction is currently running. Allows one thread to work out if an action is still outstanding in another thread.

Add `Exists()` method to `drama::ParSys` Class.

The `drama::thread::KickNotifier` class was not handling signals correctly. Resolved.

3.4.14 From DRAMA 2 version 1.79 (Jan 2018)

Add an overload of `drama::sds::Id::CreateArgCmdStruct()` which takes an `std::initializer_list` argument.

Add `drama::ParId()` constructor which takes a `drama::ParSys()` rather then a `drama::Task`, since sometimes you have that available rather then a `Task`, and it does make sense that you have a `ParSys` available.

3.4.15 From DRAMA 2 version 1.78 (Nov 2017)

Add `git::Path::GetSimulation()` method, which returns details about the simulation setting which was applied.

Add `gitarg::Filename class`. This is a sub-class of `gitarg::String` specialized for the names of files which must exist. In particular, it checks if the file exists and can be read.

Add `gitarg::ArgFlags class`. This class is used to check for the existence of one or more flags in an SDS command argument structure. Here flags are defined keywords which must exist in an SDS command argument structure. For example, an interface might define that it might accept the keywords `DARK` and `RECORD`. The idea is that these are optional and we just want logical value set if they are present. The result will be a mask of bits representing each flag which is set. An example of using this is shown below.

```
/*
 * These are flags used for arguments.
 */
 const unsigned int OPEN=1;
 const unsigned int DARK=2;
/*
 * We need to set up a relationship between the above integer values and
strings.
 */
 using MyArgFlagType = drama::gitarg::ArgFlags<drama::gitarg::ArgFlagValVector>;
 drama::gitarg::ArgFlagValVector flagVals = { {"OPEN", OPEN }, { "DARK", DARK}
};
/*
 * We get in the constructor.  .
 */
 MyArgFlagType flagsArg(flagVals2, GetEntry().Argument());
/*
 * GetVale() returns the value (as an unsigned value).
 */
 MessageUser(std::string("flagsArg value = ") +
std::to_string(flagsArg.GetVal()));
```

3.5 DRAMA 2 - New methods for creation of single dimensional SDS arrays

From DRAMA 2 Version 1.74

New overloads have been added for the `drama::ads::Id::CreateChildArray()` and `drama::sds::Id::CreateTopLevelArray()` methods. These methods make it simpler to create a single dimensional array. These take an integer as the third argument rather then a container. E.g.

```
auto id = drama::sds::Id::CreateTopLevelArray("ArrayName", INT32, nElements);
```

Basically - you don't have to create the container yourself.

3.6 DRAMA 2 - new drama::ArrayParameter class.

From DRAMA 2 Version 1.74 The `drama::ArrayParameter` class can be used for creating parameters in the style of the "drama:Parameter" class where the parameter is a single dimensional array of scalar or string items. Previously you would have had to create these items using SDS yourself, but they are common in AAO instruments.For example, you would declare one of these like this:

```
drama::ArrayParameter<INT32> arrayParam;
drama::ArrayParameter<std::string> strArrayParam;
```

Having declared them you would construct them like this

```
arrayParam(TaskPtr(), "ARRAY_PARAM", 5, 0)
strArrayParam(TaskPtr(), "STR_ARR_PARAM", 6)
```

For the arrayParam, the "5" is the number of elements, similarly for `strArrayParam`, "6" is the number of elements. The extra argument to `arrayParam` is the initial value. For `strArrayParam`, there are two optional arguments, the length of each string (defaults to 100) and the initial value (defaults to an empty string).

Having constructed `arrayParam`, you can use it like this.

```
std::vector<INT32> intVec { 1, 2 , 3};
arrayParam.Set(intVec);
auto arrayVal = arrayParam.Get();
std::cerr << "Integer Array Parameter Values:";
for (auto a : arrayVal)
{
   std::cerr << a << " ";
}
std::cerr << std::endl;
std::cerr << "Integer Array parameter size is:"
          << arrayParam.Size()
          << std::endl;
arrayParam.Set(2,10);
std::cerr << " Item of index 2 is:"
          << arrayParam.Get(2)
          << std::endl;

std::cerr << " Item of index 2 via index operator:"
          << arrayParam[2]
          << std::endl << std::endl;
```

The `arrayParam` approach can be used for all scalar types. Note that you can't assign to `arrayParam[n]`, you can only read it. For `strParam`, it is very similar:

```
std::vector<std::string> strVec { "one", "two" , "three"};
strArrayParam.Set(strVec);
auto strArrayVal = strArrayParam.Get();
std::cerr << "String Array Parameter Values:";
for (auto a : strArrayVal)
{
   std::cerr << a << " ";
}
std::cerr << std::endl;
std::cerr << "String Array parameter size is:"
```

```
            << strArrayParam.Size()
           << ", string length is:"
            << strArrayParam.Strlen()
            << std::endl;
    strArrayParam.Set(2,"ten");
    std::cerr << " Item of index 2 is:"
            << strArrayParam.Get(2)
            << std::endl;
    std::cerr << " Item of index 2 via index operator:"
            << strArrayParam[2]
           << std::endl;
```

The only real difference in `strArrayParam` is that there is the extra "`Strlen()`" method to return the length of each string in the array.

4 DRAMA 2 Reporting/Logging Notes

Here are some details on how to report to the user and work with log files in DRAMA 2.

It should be remembered that the main sources of info about DRAMA 2 are the DRAMA 2 Book (Drama2Examples ACMM sub-systems) and the HTML pages generated form the sources by Doxygen. Also see the examples in the Drama2Examples ACMM sub-system.

4.1 General DRAMA policy on reporting/logging

A DRAMA task should ensure general (non-error/warning) messages intended for the user are send as DRAMA informational (MsgOut()) messages. Messages intended for the user but as errors or warnings should be sent via Ers. Note that DRAMA 2 exceptions will be converted to Ers if thrown out of an action handler.

Log messages should be written using the DRAMA Logging features, ensuring they appear in a well defined log file.

A delivered task, should NOT, in general, output messages to stdout or stderr, as these are often lost (not seen by the user and not logged). Only use these as part of debugging etc when developing a task, plan to remove them before delivery!

4.2 Informational Messages to the user.

In the DRAMA C interface, these are done using `MsgOut()`. This function returns a message to the user interface that initiated the action. This might be ditscmd or some purpose built GUI. Even if there is a number of tasks involved in starting your action, the message still ends up with the user interface that started things.

In DRAMA 2, the main interface to sending this messages is the `MessageUser()` method, which takes a `std::string` argument. This is a method of both `drama::MessageHandler` (non-thread actions), `drama::thread::TAction` (threaded action).

For non-threaded actions, you simply invoke this in your "`MessageReceived`" method. Similarly, for threaded actions, you invoke this within your "`ActionThread`" method.

`MessageUser()` takes a std::string argument which can include new-line characters, These are used to split the string into individual `MsgOut()` messages. This means you can create mutiline messages by, for example, creating an object of type `std::ostringstream`, for example:

```
std::ostringstream mess;
mess << "First line of message"  << std::endl
     << "Second line of message";
MessageUser(mess.str());
```

An alterative to this is particularly useful if invoking a function which takes an `std::ostream` as an argument. The `drama::MesageUserStreamBuf` class provides a way of generating a stream buffer which output via `MessageUser()`. Unlike creating the ostringstream example above, when done this way, the message is generate each time

a new line is output. This ensures the timing of the message output is correct. You create an object of type `MessageUserStreamBuf` and then use that object to initialise an `std::ostream`, too which you write your message.

```
    /* Create the stream buffer  and then a stream using it. "*this" is a
MessageHandler or TAction class */
    drama::MessageUserStreamBuf<decltype(*this)> messStreamBuf(*this);
    std::ostream messStream(&messStreamBuf);
    messStream << "First line of message"  << std::endl;
    messStream << "Second line of message" << std::endl
    messStream <<  "This line output when messStream destructor is run, since no end
of line follows";
```

The template argument to StreamBuf would normally be a drama::MessageHander (non-threaded action) or drama::thread::TAction (threaded action) but can actually be any class which has a MessageUser() method. The template argument `decltype(*this)` is just used to I have the right type.

The DRAMA2 test of this is my simplest example of this - see `DRAMA2/examples/msgoutstream.cpp`. Class/method documentation in the header file `$DRAMA2_DIR/drama/messagehandler.hh`

4.3 If you want to send error reports without throwing exceptions.

Most errors in DRAMA2 programs should result in a `drama::Exception()` being throwm, but sometimes you want to send error information without ending the action. In the DRAMA C interface, this is typically done by using `ErsRep()` to report formatted lines of text, and then `ErsFlush()` to cause the message to go to the user (with `ErsOut()` just being an `ErsRep()` followed by `ErsFlush()`). This would particularly be the case for warnings! (And warnings should use Ers rather then `MessageUser()`, since a GUI can format/display them to bring them to the attention of the user)

In DRAMA 2, the issue is that ERS uses a global variable and its use is therefore not thread safe. It is quite ok to use in a non-threaded action, but in the threaded action you must do something like this:

```
{
  drama::thread::AccessDrama dramaLock(*tMessHandler);
  StatusType status = MY_STATUS_CODE;
  drama::ErsReport("My warning report", &status);
  ErsFlush(&status);
}
```

Where `tMessHandler` is an object of type `drama::thread::TAction`, the threaded action object. The construction of "`dramaLock`" will take the DRAMA 2 Lock, and also ensure the internal DRAMA details are correct for your action. When dramaLock is destroy, the internal DRAMA details restored before the lock is released.

`ErsReport()` is a simplified version of `ErsRep()`. It can take a std::string with multiple lines Separated by newline (\n) characters, and will call `ErsRep()` for each line.

A more typical use of `ErsReport()` might be to write your message to string stream and then report it. This for example is how I converted an exception to a warning using this function.

```
catch (const drama::Exception &e)
{
  std::ostringstream errStream;
  errStream << "POLL to task "
            << GetTaskName()
            << " failed with DRAMA exception:"
            << std::endl;
  // Status for Ers calls should e form the exception.
  StatusType status = e.dramaStatus();
 // Report errString via ERS.
  drama::ErsReport(errStream.str(), &status);
```

```
    // Report the details of the exception via Ers.
    e.ErsRep();
    # Flush ERS report to the user.
    ErsFlush(&status);
  }
```

The dramaLock wasn't needed here as it was already taken.

4.4 Logging in a C++ compatible way

(Note - also see below for alternative formatting)For logging, in addition to the traditional `Log()` method which takes a C `printf()` style argument, there is also now an `SLog()` method and a `LogStreamBuf` class which give a more C++ interface to logging.`SLog()` looks a bit like `Log()`, but instead of using C-Printf style formatting, it uses a more C++ approach, compatible with all types which can be output to a stream. The format string has a simple "%" character (not, for example, a "%d", just the "%") at each point where an argument is to be inserted. So rather then doing this:

```
logger.Log(D2LOG_INST, false,"LogTst:ActionThread",
           "This = %p:Testing logger Log() method, test string = %s",
           (void *)this, "some string #1");
```

You might do this:

```
logger.SLog(D2LOG_INST, "LogTst:ActionThread",
            "This = %:Testing logger SLog() method, test string = %",
            (void *)this, std::string("some string #2"));
```

The `drama::logging::LogStreamBuf` class allows you to generate a stream to which you can log. Each time a new line (std::endl) is sent to the stream, the stream is flushed to the log file. You create an object of type `LogStreamBuf`, and then use that to initialise an `std::stream` to which you write your messages. For example:

```
drama::logging::LogStreamBuf logBuf(&logger,
                                    D2LOG_INST,
                                    "LogTst:ActionThreadS");

std::ostream slogger(&logBuf);
slogger << "Slogger:First line of logging output, target task:"
        << std::setw(10)
        << _targetTask
        << std::endl;
slogger << "SLogger:Second line of logging output" << std::endl;
```

See `$DRAMA2_DIR/drama/logger.hh` for class/method documentation.There is an additional version of `SLog()`. It takes an additional argument, BEFORE the others. This is a stream which gets a copy of the message is sent to the log. So I typically start off doing something like this whilst developing a program:

```
SLog(std::cerr, D2LOG_INST, ...);
```

This sends the log message to `stderr` as well as the log file. When I am happy and no longer want this spilling on stderr, I remove the "`std::cerr,`" and now it just goes to the log file.

4.5 Logging in your own threads

If starting your own threads within your threaded actions, you should tell the logging system about it. What you are doing here is associating the action, and the thread. The logging code will use the thread id to find the action when it logs a message.The lines here would normally best part of the first few lines in your thread. You do need to have available the address of `drama::thread::TAction` object which is implementing your threaded action, `tMessHandler` above.This is typical call to access the logger from a drama::Task pointer.

```
drama::logging::Logger &myLogger = thisTask→Logger();
```

Then you can register your thread.

```
myLogger.RegisterThread(tMessHandler->GetMessageContext(),  // Associate this action
                        std::this_thread::get_id(),         // with this thread.
                        "PathIter:Thread");                 // Name of thread
```

The thread name argument ("`PathIter:Thread`") appears in the logger, in the "`ThreadFunc`" column. The idea is that it is the name of the function implementing the thread, but it can be anything that means something to the reader of the log.

4.6 Safe C++ print in C printf style.

First note that you would normally use one of the features above to output messages, but sometimes you just want to send something to `std:cerr` or `std::cout` or similar, particularly whilst building your task.In C++ it is recommend that you use streams to format your output rather then the C printf() style functions. This ensures you can output class objects correctly. But, the standard stream approach can quickly get unwieldy. Something like:

```
printf("Arg 1:%s, Arg 2:%s, Arg 3:%s\n", argv[1], argv[2], argv[3]);
```

Becomes:

```
std::cout << "Arg 1:" << argv[1] << ", Arg 2:" << argv[2], ", Arg 3:" << argv[3] << std::endl;
```

Which is both longer and harder to read. And this is a simple case! The web is full of questions from people asking if there is a C++ equivalent which is type safe. Fortunately, whilst no standard function is provided (before C++20), C++11 does allow you do do something pretty close this and DRAMA provides a couple of implementations (including the "`SLog()`" method of the logger, above). These approaches use C++ variable argument list features combined with an `ostream`.DRAMA2 Provides two such methods for output to any stream, these are `drama::SafePrintf()` and `drama::TSafePrintf()`. Both of these take a stream argument, a format string and a list of arguments for the format. The format string has a simple "%" character (not, for example, a "%d", just the "%") at each point where an argument is to be inserted. For example:

```
drama::SafePrintf(std::cerr, "Initialise completed for task %\n", targetTask->GetTaskName());
```

Any number of arguments are allowed and any argument which can be written to a stream will work. The main flaw compared to direct use of streams or `printf()` is that failing to supply the correct number of arguments is not detected until run time. `SafePrintf()` is more efficient then `TSafePrintf()`, but the former is not thread safe (a single line of output to a file could be split due to output from another thread to the same file). `TSafePrintf()` first formats to a `std::ostringstream`, which is then written to the user's supplied stream in one operation. Less efficient but thread safe. C++20 does support an alternative approach to formatting, which can now be used in C++11/C++17 DRAMA programs - see below. But you will still want to use the drama::SafePrintf() function with this.

Another problem with this approach (that may move you back to direct use of streams) is less control over formatting of individual arguments. There are no formatting specifiers which can be associated with the % character. Instead you must apply stream I/O manipulators to format before the call or as extra arguments. For example

```
std::cerr << std::fixed << std::showpoint;
drama::SafePrintf(std::cerr, "Testing of output via SafePrintf - %,% %\n",
                  "string", std::setprecision(2), 12.3333);
```

Here, before invoking `SafePrintf()`, I changed the stream formatting of floating points to fixed format. In the argument to `SafePrintf()` I supplied an extra "%" character (the second one), the corresonding argument of which indicates the precision for the next argument. This approach does work, but does restrict your argument string as two "%" characters in a row are used to cause one "%" to be output, so there is now way to set the precision without an extra character between that "%" and the next one.

4.7 Formatting Strings - C++20 style

C++20 introduced the std::format class. This is actually a much better way in most cases. If you task uses C++20 or later, then use that to format a string for output.

For older C++11 and C++17, DRAMA provides a version of this (which will work with later C++ but may end up out of date in minor ways).

Instead of including <format>, you include <drama/fmt/format.h>. Instead of specifying "std::format", you specify "fmt::format". Other then those changes, it can be used in the same way.

The example below shows many uses of this (from drama_source/Drama2/examples/msgoutfmt.cpp).

```
    drama::Request MessageReceived() {
        /*
         * Send a message to the user.  (Method of MessageHandler)
         */
        MessageUser("Starting...");

        /* Basic hello world for the fmt library */
        MessageUser(fmt::format("The answer is {}", 42));

        /* Demonstrating referencing argument by number */
        MessageUser(fmt::format("Ref args by number:{0}, {1}, {2}, repeat of arg 1 {0}",
"arg 1", "arg 2", "arg 3"));

        /* Demonstrate named arguments */
        MessageUser(fmt::format("Hello, {name}! The answer is {number}. Goodbye, {name}.",
           fmt::arg("name", "World"), fmt::arg("number", 42)));

        MessageUser(fmt::format("{:<40}", "left aligned"));
        MessageUser(fmt::format("{:>40}", "right aligned"));
        MessageUser(fmt::format("{:^40}", "centered"));
        MessageUser(fmt::format("{:*^40}", "centered with * as fill"));  // use '*' as a
fill char

        // Dynamic with
        MessageUser(fmt::format("{:<{}}", "left aligned - dynamic with", 40));
        // Dynamic precision
        MessageUser(fmt::format("{:.{}f}", 3.14, 1));

        MessageUser(fmt::format("int: {0:d};  hex: {0:x};  oct: {0:o}; bin: {0:b}", 42));
        // Result: "int: 42;  hex: 2a;  oct: 52; bin: 101010"
        // with 0x or 0 or 0b as prefix:
        MessageUser(fmt::format("int: {0:d};  hex: {0:#x};  oct: {0:#o};  bin: {0:#b}",
42));
        // Result: "int: 42;  hex: 0x2a;  oct: 052;  bin: 0b101010"

        auto t = tm();
        t.tm_year = 2010 - 1900;
        t.tm_mon = 7;
        t.tm_mday = 4;
        t.tm_hour = 12;
        t.tm_min = 15;
        t.tm_sec = 58;
        MessageUser(fmt::format("{:%Y-%m-%d %H:%M:%S}", t));
        // Prints: 2010-08-04 12:15:58

        MessageUser("Finished");
        /*
         * Tell DRAMA the action has ended.
         */
        return  drama::RequestCode::End;
    }
```

Code Block 1 Using fmt::format

5 DRAMA 2 - integrating C++ streams into DRAMA messages.

This page gives some examples of how to integrate the use of C++ streams into DRAMA 2 tasks. In particular, I am thinking of libraries etc. which use C++ streams for reporting, but you need to transmit the results as DRAMA Error (ERS) or Informational (MsgOut(), MessageUser()) messages to ensure they get back to the user interface which initiated the action.

5.1 Error Reports

DRAMA Error reporting is via the Ers system. It is easy in C++ to default to reporting messages to std::cerr, but such messages are not correctly delivered to the user in DRAMA systems. So how do we create a general purpose class/library call which works in both cases, particularly if we want to take advantage of C++ streams.

One approach which works well is to have your method/function accept an extra argument, which defaults to `std::cerr`. For example

```
// Check if a position is currently observable on the UKST. Returns true if it
is.
 bool CheckObservable(double RADeg, double DecDeg,
                      std::ostream &errStream=std::cerr);
```

The function will use that third argument as the destination of its error report details.

If called without the third argument, it defaults to `std::cerr`. But you can call it such as to capture that output, and then output it via ERS using an `std::ostringstream`. For example:

```
std::ostringstream errStream;
if (!ukst::CheckObservable(RA, Dec, errStream))
{
    drama::Task::guardType dramaLock(Lock()); // Needed for ERS operation
    StatusType status = TAIPANTCS__INVPOS;
    drama::ErsReport(errStream.str(), &status);
    DramaTHROW_S(status, "Position (%, %) is not observable at this time", RA,
Dec);
}
```

Here if the call fails, we are reporting the details via ERS before throwing an exception. The result will that all information gets back to the user.

Note that you must take the DRAMA task lock before invoking `drama::ErsReport()`. That function takes a `std::string` as its first argument, which you can easily get out of the string stream. Additionally, it will write an ERS report line for each line in the supplied string. So the report within the the stream gets broken back into lines before it is delivered to the user.

It must be noted that in the above case, nothing gets reported to the user unless the function returns false.

5.2 Informational Messages

Informational messages can be done in a similar way to ERS messages - e.g. using an `std::ostringstream` to record the required information and then using `MessageUser()` to get it back to the user, but this will rarely work as required - since messages are not sent to the user until you call `MessageUser()`. In the ERS case, you probably don't want any reports to the user unless things fail, but for informational messages, you want them regardless and you want them to go out at the time they occur so you can see when they actually occur.

You can use `drama::MessageUserStreamBuf` to get the desired effect. This is used to create a `std::ostream` which sends a message to the user via MessageUser() each time a `std::endl` is written to the stream, or whenever the stream is synced.

For example, consider this method

```
virtual void DomeOpen(std::ostream &logStream=std::cerr) = 0;
```

This method will use the argument as the destination of any logging messages, and by default, will use to `std::cerr`. The code below shows how I would log them via `MessageUser()` such that they are sent immediately they occur.:

```
// First create a MessageUserStreamBuf item using the action (actually, anything
with a MessageUser() method can replace "*threadAct")
 drama::MessageUserStreamBuf<decltype(*threadAct)> messStreamBuf(*threadAct);
```

```
// Then create a stream using this buffer.
 std::ostream messStream(&messStreamBuf);
// Then make the required call.  Anything written to the stream will be sent
using threadAct->MessageUser().
 _telescopeIF->DomeOpen(messStream);
```

The key here is that the `MesageUserStreamBuf` expects its template type to support a `MessageUser()` method, which is invoked each time a `std::endl` is written to the stream, or each time it is synced. Objects of both the non-thread action type (`drama::MessageHandler`) and threaded actions (`drama::thread::TAction`) and any sub-classes can be used as the template argument to `MesageUserStreamBuf`.

5.3 Other relevant information

DRAMA 2 Reporting/Logging Notes

DRAMA 2 programs and exceptions.

6 DRAMA 2 - ensuring ERS, "gitarg" classes, and ERS from old libraries work from threads.

The DRAMA ERS system is the one internal bit of DRAMA that uses a global variable that is not controlled by the DRAMA 2 lock (at least in the current implementation). As a result, use of ERS from threads can cause corruption and crashes.

As a result, to ensure no problems occur, threads must take the DRAMA lock if using ERS. Ideally, you will do this using a drama::thread::AccessDrama item. That item takes the DRAMA lock but does so in a way that ensures any ERS output is delivered to the correct parent action. If calling DRAMA within an action thread, the argument to the constructor will be a/or a sub-class of drama::thread::TAction. If in a UFACE thread, it will be a/or a sub-class of drama::thread::TUface. (both TAction and UFace are sub-classes of drama::thread::TMessageHandler, which is what the argument needs to be)

You want to take the lock for as short a time as possible, and you must ensure you release it even if your code throws an exception, so it is normal to put the relevant code in its own code block. E.g.

```
{
   StatusType status = XXX__MY_STATUS_OK;
   drama::thread::AccessDrama dramaLock(*this);
   ErsReport("Some error report I want to make", &status);

}
```

In the above code, "*this" is a sub-class of `drama::thread::TAction`.It is also important to deal with ERS messages before you give up the lock. E.g. Flush/Annul them or add them to an drama::Exception report.

6.1 gitarg classes use ERS when getting argument values.

It is important to note that the "gitarg" classes use ERS when fetching the parameter value. As a result, these calls, when made from a thread, must also be done within a DRAMA lock. This does add the complication that if you need the value outside of the code block where the lock is taken, you must construct the variables outside and then use their Get() methods, rather then trying using the version of the constructor that does an implicit `Get()`. For example:

```
drama::gitarg::String resetTasks = "ALL";
drama::git::ResetType resetType(drama::git::ResetEnum::Recover);
```

```
{
  drama::thread::AccessDrama dramaLock(*this);
  resetType.Get(obeyArg);
  resetTasks.Get(obeyArg, "TASKS", 2, "ALL", drama::gitarg::Flags::NoFlagSet);
}
```

The git classes will throw an exception if they fail and (from DRAMA2 version 1.81) will ensure any ERS reports current at that point are added to the exception details. (Prior to 1.81, GIT was inconsistent about this).

6.2 Other possible areas with this issue.

ERS has been used extensively through older DRAMA code. DRAMA2's problem with gitarg is due to its use of the GitArgGet series of C language interfaces. Some of the DRAMA libraries built on GIT, e.g. DUL and the TCL interfaces are also likely to have this issue.

6.3 Using a raw DRAMA lock instead of AccessDrama

If at all possible, you should use the AccessDrama class to task the DRAMA lock. You are likely to appear to get away with taking the raw DRAMA lock in many cases - but because the DRAMA action context is not set up correctly, the resulting messages may not be sent to the correct parent action.

7 DRAMA refresher course topics.

This page is being used to record some details of what should be in DRAMA refresher course that I am intending to run. People are free to add comments/edits to indicate which should be included.

7.1 DRAMA Systems overview

The general task layout stuff. Taken from DramaTalk and DramaExample - used for the older introductory talks

7.2 Error handling

7.2.1 Error/Status codes

The MESSGEN utility and the Mess routines provide a portable technique for generating unique error codes and then associating text with the error code. MESSGEN can generate?? include files defining constants for each error in C, Fortran?? ??VAX Pascal, TCL and JAVA.

The Mess routines provide the ability to fetch the text associated with each message at run time??.

There is a "Facility" code - which defines a set of status codes. These are defined by a file kept in ACMM. When you create a DRAMA sub-system in ACMM,

why/how to create them/how to translate them. Using them in DRAMA 1 code. Using them in DRAMA 2 code.

7.2.2 Ers

- who/how. Intergrations with DRAMA 2.

7.2.3 Exceptions

- both with DRAMA 2 and without. How to handle exceptions generated by other packages (not sub-classes of drama::Exception)

7.3 General Messages to the user

DRAMA Supports a simple message for sending informational messages to the user. This is generally known as the "MsgOut" approach. An action can invoke MsgOut() to format a message (C printf style formatting) and send it to the originator of the action. Limit of 200 characters.

For example, in 2dF, if the Gripper task wants to send a message to the user, it just invoked MsgOut(). The message is sent to the Positioner Task, which sends it back to the control task.

Since the Control Task is a GUI it was probably the user interface that started the action. DRAMA will be default spill the message on stdout, but allows a task to specify a handler routine (DitsUfacePutMsgOut()), which is how I put these messages in a scrolling message area.

By convention, any MsgOut() message which starts with "WARNING:" is formatted in a way that highlights it.

In DRAMA2, the method MessageUser() is available in action handlers. It does the same thing but using modern C++ approaches.

7.4 DITSCMD

Working with ditscmd to determine what is happening. In truth, you don't need ditscmd, any flexible GUI will do, but ditscmd has been my quick tool (think "vi" equivalent) so has picked up a few extras

- ditscmd -h
  Gives you all the details
- `ditscmd -o / -k`

`Send Obey/Kick messages.  -o is the default.`

`ditscmd TICKER TICK`

`ditscmd -k TICKER TICK`

- ditscmd -g Multiple parameters, within complex parameters
  Special parameter names - `_NAMES_` and `_ALL_` to fetch all parameter names and all parameters.
  Often good to combine with "-v" - verbose mode

```
>> ditscmd -gv TICKER _ALL_
DITSCMD_1c95:SdpStructure StructDITSCMD_1c95: PARAM1 Int 1DITSCMD_1c95: PARAM2
Char [9] "hi there"DITSCMD_1c95: PARAM3 Float 2.2DITSCMD_1c95: PARAM4 UInt
3DITSCMD_1c95: STRUCT_PARAM StructDITSCMD_1c95: STRUCT_VALUE_1 Int
10DITSCMD_1c95: STRUCT_VALUE_2 Int 20
```

`>> ditscmd -g TICKER STRUCT_PARAM.STRUCT_VALUE_1 PARAM1DITSCMD_1d9d:10 1`

- ditscmd -s Including setting complex parameters

  >> ditscmd -g TICKER STRUCT_PARAM.STRUCT_VALUE_1 15

- ditscmd -p How to monitor parameters

  `>> ditscmd -p TICKER START PARAM1`

- ditscmd -m:<n1>:<n2> Working with message buffer sizes
  <n1> is the size of the message you are sending. 200 should be plenty.
  <n2> is the size of the message you expect back.

- ditscmd -w

  Any returned SDS structure is written to the file ditscmd_out.sds. (No way to change the file name - program

structure issue)

- ditscmd -v

  Verbose mode. What this really means is that returned structures are output using SdsList() rather then being converted to a string. Becoming my favourite

- ditscmd -n <name>

  Set the DRAMA name for ditscmd. Useful if you don't want the name changing all the time (e.g. automated regression testing)

- ditscmd -x

  Don't complain if the returned message status indicates the task has died. Normally used when sending EXIT in scripts.

- arguments (including -z option)

  Arguments to actions are wrapped up in SDS. E.g.

  ditscmd TICKER TICK A B C

  Will send argument to the TICT action looking like (SdsList output):

```
ArgStructure Struct
       Argument1 Char [2] "A"
       Argument2 Char [2] "B"
       Argument3 Char [2] "C"
```

```
I like all commands to accept this format, but some actions done, they want named
   arguments. You can do
   ditscmd TICKER TICK ARG1=A ARG2=B ARG3=C
```

Now the argument structure looks like:

-
- ditscmd -a <file>

  Use the specified file name as the argument to the message. Allows you to construct complex arguments in other ways.

- ditscmd -h

  DITSCMD help. Explains all of this. Use often!

7.4.1 Control Messages

ditscmd -c

Internal details of a task - what actions it supports, which actions are running, what parameter monitors are active, SDS leak checking info, set debugging flags.

See DRAMA - Control messages - how to work out what a task is doing!

7.5 SDS

A library and data format for storing/moving self describing structured data around. Anything a "C" structure can represent can be represented in SDS, and additionally, arrays of structures of different types.

An SDS structure is stored in memory in a form which is efficient to work with, but can be written to/read from a byte stream. The sds file format is just the byte stream written to disk. The byte stream format is known as the "external" format. When you read one of these into memory, you can't change its shape, but you can change the values. The other format is the "internal" format. With this version, you have full control - that is how you create a structure.

Since it is self describing, generalised tools can work with them (sdslist, sdsdump, sdspoke, sdsexam).

7.5.1 Dealing with IDs in C interfaces.

When working with SDS in your code, you are dealing with a set of Identifiers (IDs) - simple integers, which refer to components of structures. A single item can have many IDs referring to it. A structured item can have IDs referring to various parts of the structure.

A top-level ID refers to the top of an SDS structure. If you lose this one, then the entire structure is lot (a memory leak!).

You create items and get an ID back. You can use that ID to carry out a lot of operations on the item (e.g. insert other items (if it is a structured item), insert this item within another, export the item to a byte stream, put values into it).

An individual SDS item may be a primitive item (char, unsigned char, short, unsigned short, INT32, UINT32, INT64, UINT64, double, float). Please use the special INT32 etc. values, as "int" and "long int" have different sizes depending on machine and compiler. It may also be an array of primitives, a structure (contains other items) or an array of structures. All arrays can have up to seven dimensions (I've never had to use that many).

Show DramaTalk example!

Note - you must tidy up correctly, otherwise you can get memory leaks and performance hits. If you create an item, you must invoke SdsDelete() on it. If you read an item from a disk (SdsRead()), you must invoked SdsReadFree() on it. All IDs must be freed using SdsFreeId(). The first two just cause memory leaks, the later can really slow down a program.

The Arg library is a set of wrappers to SDS intended for use with command arguments. Get/Put from simple structures with conversion to requested type, if possible.

7.5.2 SDS Leak issues

7.5.2.1 Memory leaks

As mentioned above, you must call SdsDelete() or SdsReadFree() on an item if you are finished with it. Otherwise you will leak memory. Particularly important in DRAMA actions which do the same thing over and over again.

7.5.2.2 ID leaks

You must also call SdsFreeId() on any ID you have generated. If you don't, then your program will slow down over time (plus have a small memory leak).

7.5.2.3 Tracking them.

Part of the documentation tidbits page shows various ways of debugging SDS leaks. Note - DRAMA itself is SDS Leak safe.

7.5.2.4 Older C++ interface.

The "SdsId" class was the original C++ interface to SDS. The important addition is the use of destructors to tidy up. Objects of these types are using in much of C++ code up until Taipan (about 2015) when DRAMA2 arrived.

It does not use exceptions, still has status arguments.

Basic approach is that the class object knows if SdsDelete(), SdsFreeId() or SdsReadFree() should be invoked when the destructor is run. Hence you "normally" don't need to worry about these things (normally - but if importing/exporting from/to C, you do).

No copy/assignment operators (what do you really want to do if you copy/assign, unclear). There are methods to allow you to do this, but you are forced to think harder. Lots of passing the objects around by pointer.

Construction of objects using a set of constructors with different arguments. Separate Arg class provides the Arg style wrappers.

7.5.3 DRAMA 2 Interface.

The drama::sds::Id class. Combines Arg and SDS into one and has a much expanded interface (array access, better type safety etc). Uses a bunch of factory constructors to create items rather then normal constructors, e.g. CreateTopLevel(), CreateTopLevelArray(), to make it clearer what you are doing.

Still not copy/assignment operator, but you do get "move copy" and "move assignment" (C++11 feature). These handle most cases of interest and allow the factory constructors to work.

Uses exceptions rather then status arguments.

The way I write this stuff is often very long-handed, I often do the full "drama::sds::Id" for clarity, the the code actually does end up a lot simpler, and certainly safer.

Examples needed.

7.6 Logging

Remember that DRAMA tasks are spread across various machines. A task may be running in a spot where no stderr/stdout to the user is available. So as a habit, these should not be used for installed systems.

ERS and MsgOut() messages to get send back the the user interface which started an action.

For other stuff, the DRAMA logging system is what you need. This writes a time-tagged log file in a relatively efficient manner. You can set flags which determine what is logged, and for example, include significant amount of internal DRAMA logs.

7.6.1 Creating log file and writing log messages.

7.6.2 Location of files.

7.6.3 Management of log files and log levels.

Parameter to find name.

7.6.4 DRAMA 2 Threads

Threaded actions. Starting and managing threads. Locks, issues.

7.7 Actions and Parameters

Sometimes it is not clear if you should use an action, or use a parameter for something. E.g. telling a task configuration value, making information available to the user.

7.8 Action arguments

How to read action arguments - use of the GitArgGet series of functions and related C++ classes.

7.8.1 Older C++ interfaces

7.8.2 DRAMA 2 gitarg interfaces

7.9 Generic Instrumentation Tasks

Why/How

7.10 Message sending

7.11 DRAMA Networking system

Master task, dits_netstart

cleanup

7.11.1 IMP_Startup files

7.11.2 IMP Port numbers

7.11.3 DRAMAUNITSTART

7.12 Storing configuration files

Both read-only and read-write.

7.13 Build processes

Aims/reasoning/benefits

Specific examples of unusual stuff

AAT instrumentation software layout.

7.14 Regression testing

Simple tasks to help testing

Running tests from dmakefiles

7.15 Documentation and help

See this page: DRAMA and AAT/UKST Software Instrumentation Documentation

7.16 Bulk Data

Technique for moving very large amounts of data around.

7.17 Interfaces between DRAMA and other systems

By default, DRAMA uses the most efficient notification mechanism between two task.

IMP allows you to use a file notifier technique - chosen when you design the task.
Just a flag to DitsAppInit().

Then you can get access to the file FD.

DRAMA provides DitsAltIn() series , which can manage this FD and other systems FDs.

You can provide this FD to another system's equivalent.
You respond by processing the message.

7.17.1 DJAVA/DRAMA 2 - use a thread!

Think about the interactions between the two (are locks required)

7.18 Finding files.

7.18.1 Where should they be?

Things that are read only, not changed by tech staff

> *$<subsys>_DIR*

7.18.2 IRIS 2 spectrograph task has a configuration file, filters on a wheel. Things that are read/write.

Think about it a bit.
User specific? Probably in $HOME or somewhere from $HOME.
Not user specific: More general application question

Things that are normally read only by the task, but may want to be modified by staff
E.g. lamp details.

Typically.

First look in $HOME
Then look in $<subsys>_DIR

Think about that is safe to be changed by technical staff/astronomers
May need to have two files? or some other validation.

Maybe an instrument specific directory structure
E.g. 2dF positioner directory,
Taipan configuration files

We haven't always done this right. E.g.
/instsoft/2dF/config -> has lots of bits.

7.18.3 Always find things via environment variables!!!

getenv() / DitsGetSymbol()

DitsGetSymbol() is more portable (e.g. to VMS/Windows) but not as necessary.

Dul series of routines:

DulFindFile()

Find a file using search paths, translating environment variables.

Followed techniques used on VMS.
DITS_DIR:SDS_DIR:

HOME:<subsys_dir>:

.:<subsys_dir>

Needs a proper C++ interface

DulParseFileName()

Note - ImpMaster is limited to 63 characters in a file name.
Workaround, use format
DITS_DEV:ticker

(Consider fixing???)

8 The SDS Compiler - creating/using SDS structures matching C structures - with DRAMA2

One of the issues in SDS is that we often want to create quite complex structures in SDS, which match complex C structures. The low level SDS Put/Get operations provide easy ways to move data between C structures and SDS, and this includes complex structures. But how do you ensure your SDS structure is a correct equivalent of your C structure!

The "SDS Complier" - `sdsc`, is the tool you need. Take for example the following C structure that I want to move to/from SDS.

```
#define ARRAY_LEN 10
```

```
typedef INT32 myIntType;
typedef struct subStruct {
   INT32 anInt;
} subStructType;
struct myStruct {
    myIntType anInt32Bit;
    char aChar;
   unsigned short aShortArray[ARRAY_LEN];
  unsigned char secondChar;
  subStructType nestedStruct;
};
```

If you have such a C structure and an equivalent SDS structure, you can move it easily between C and SDS, for example:

```
/* Create the SDS structure - some how! */
drama::sds::Id id = ...
/* Create a C equivalent structure. */
myStruct theStruct;
/* Put some stuff into the C structure. */
theStruct.anInt32Bit = 1;
theStruct.aChar = 'a';
for (unsigned i = 0; i < ARRAY_LEN ; ++i) theStruct.aShortArray[i] = i*10;
theStruct.secondChar = 'b';
theStruct.nestedStruct.anInt = 20;
/* And then put the result in the SDS structure. */
id.Put(sizeof(theStruct), &theStruct);
/* And list it to see what we did on stdout. */
id.List();
/* Retrieve it into another version of myStruct. */
myStruct theStruct2; id.Get(sizeof(theStruct2), &theStruct2);
/* Check a few of the values to confirm we read it ok. */
assert(theStruct.anInt32Bit == theStruct2.anInt32Bit);
assert(theStruct.secondChar == theStruct2.secondChar);
assert(theStruct.nestedStruct.anInt == theStruct2.nestedStruct.anInt);
```

The above code first creates the SDS structure - something we are ignoring for the moment. It then declares a variable of the right type - "`myStruct`", named "`theStruct`". It then fills it out with some data. Once this is done, the `id.Put()` method can be used to put the data into the SDS structure. And later, we can get it out again using `id.Get()`.

If you run the above program, you expect to see the output by the `id.List()` method.

```
myStruct        Struct
 anInt32Bit       Int    1
 aChar           Char   "a"
 aShortArray       Ushort [10] {0,10,20,30,40,50,60,70,80,90}
 secondChar        Ubyte  98
 nestedStruct      Struct
  anInt            Int    20
```

Which if you look back at the code and the structure, makes sense.

It must be noted that, whilst the above code is C++, the structure itself is more a C structure - a POD (Plain old data) structure. It can't have methods and can't have members that have methods. There are of course many ways of wrapping such items in types with methods, but the bare type that moves between C/C++ and SDS must be a POD type.

8.1 Creating the SDS Structure

If you just launch into C/C++ calls to create the SDS structure, it can quickly get out of control, and, in particular, you might have trouble ensuring the C and SDS structures are equivalent, particularly as things change. The solution is to use the "`sdsc`" tool, the "SDS Compiler". This can read a C language (not C++) .h file and writes a C function which generates an equivalent SDS structure.

Put your structure definition in a simple include file - try to avoid including any other include files, since they may bring in C++ code and other things like function prototypes which are not supported by "`sdsc`". Then you need to define an item of the type of interest (create a variable of that type). One thing to help here is that the macro `___SDS___` (three underscores

each side) is defined when "sdsc" is being run, so you can make your include file look a little different under "sdsc" compared to running it through the C compiler. This is what I find for the example above (after declaring "myStruct").

```
#ifdef ___SDS___
 struct myStruct theStruct;
#endif
```

Having done this, I can now compile this include file (which I named sds_example.h) to generate a C function. In this example, I want to name the C structure "CreateStruct" and have it written to the file "sdsc_ex_cre_func.c". This is the command I use:

```
$SDS_DEV/sdsc -f CreateStruct -tmyStruct -NtheStruct sdsc_example.h sdsc_ex_cre_func.c
```

Here - the "-f" option specifies the name of the function to create (CreateStruct), the -t option the structure type (myStruct), the -N option the name of the variable of that structure type (theStruct). After the options, the first argument is the input include file (sdsc_example.h) and the second is output file (sdsc_ex_cre_func.c).

"sdsc" will run the C pre-processor over the file before compiling it, so you can use C pre-processor macros as required.

The result is that the file sdsc_ex_cre_func.c will be created containing

```
#define NULL 0
 #include "sds.h"
 extern SdsIdType CreateStruct (StatusType *status) {
/* Definition of structure "theStruct" of type "myStruct" at level 0 (sdsc/sdsc_example.h:33)*/ SdsIdType tid0 = 0; /*
SDS id for structure*/
 unsigned long dims[1]; /* Dimension array for array elements*/
 SdsIdType id = 0; /* Primitive element sds id */
if (*status != STATUS__OK) return(0);
SdsNew(0,"myStruct", 0 , NULL, SDS_STRUCT, 0, NULL, &tid0, status);
SdsNew(tid0, "anInt32Bit", 0, NULL, SDS_INT, 0 , NULL, &id, status); SdsFreeId(id,status);
 SdsNew(tid0, "aChar", 0, NULL, SDS_CHAR, 0 , NULL, &id, status); SdsFreeId(id,status);
dims[0] = 10;
 SdsNew(tid0, "aShortArray", 0, NULL, SDS_USHORT, 1 , dims, &id, status); SdsFreeId(id,status);
 SdsNew(tid0, "secondChar", 0, NULL, SDS_UBYTE, 0 , NULL, &id, status); SdsFreeId(id,status);
 {
   /* Definition of structure "nestedStruct" of type "subStruct" at level 1 (sdsc/sdsc_example.h:29)*/
   SdsIdType tid1 = 0; /* SDS id for structure*/
    SdsNew(tid0,"nestedStruct", 0 , NULL, SDS_STRUCT, 0, NULL, &tid1, status);
    SdsNew(tid1, "anInt", 0, NULL, SDS_INT, 0 , NULL, &id, status);
   SdsFreeId(id,status); SdsFreeId(tid1,status);
    /* End of structure definition at level 1 */
}
/* End of structure definition at level 0 */
return(tid0); }
```

And all you need to do invoke CreateStruct() to create the SDS structure, and compile and link sdsc_ex_cre_func.c into your program. By using different names, you can use this feature many times in the one program.

8.1.1 Running sdsc from a dmakefile

To run the sdsc compiler from a dmakefile, use the SdsIncludeFunction() macro. Its arguments are the destination file, the function name, the structure type, the structure name and the include file. The following example is what actually generated the command I used above.

```
SdsIncludeFunction(sdsc_ex_cre_func.c, CreateStruct,myStruct,theStruct,
sdsc_example.h )
```

The example code can be built using a dmakefile containing these lines:

```
INCLUDES =  -I. DramaIncl IDir(DRAMA2_DIR)
USERCCCOPTIONS = AnsiCC11Full()
LIBS=LinkLibDir(DRAMA2_LIB,drama2)          /* Libraries to link against */
```

```
NormalCRules()              /* Include normal c rules      */
all : sdsc_example
DramaCPPProgramTarget(sdsc_example, sdsc_example.o sdsc_ex_cre_func.o, ,$(LIBS) ,)
SdsIncludeFunction(sdsc_ex_cre_func.c, CreateStruct,myStruct,theStruct, sdsc_example.h )
```

8.1.2 Example files

The example files are all maintained in the Drama2Examples ACMM sub-system, within the directory `sdsc`. They are attached for reference.

sdsc_example.hsdsc_example.cppsdsc_ex_cre_func.c

dmakefile

Note - all of this was tested with SDS ACMM version 3.87 and Drama2Examples version

8.1.3 Older code using sdsc

The "`sdsc`" program originally created the C function without giving it a name, with the intention being that you then included the relevant code inside a C file which provided the function name. This is rather complicated but is done in various bits of AAO code, the 2dF control task being one example. When run this easy from a dmakefile, the macro "SdsIncludeFile" is used to run the program.

"sdsc" originally allowed you to use C integer types - "int", "long int" an related unsigned types. This is no longer possible as the translation between these types and SDS is ambiguous. You instead need to use the `INT32`, `UINT32`, `INT64` and `UINT64` types. The old approach is very likely to get into trouble in 64 bit machines.

8.2 dmkmf/dmakefile tricks and treats

This command is meant to explain some of less obvious features of the dmkmf command and dmakefile's.

8.2.1 make release

8.2.2 make enable

8.2.3 make dramadirs

8.2.4 make dramadirsl

8.2.5 make dramadirs2

8.2.6 make dramadirs2l

8.2.7 dmkmf -g

8.2.8 dmkmf -O

8.2.9 dmkmf -v <xxx>

Version_<xxx>

8.2.10 dmkmf -noautotest

8.2.11 Testing from dmakefile's.

```
DummyTarget(All, checktarget includes Lib(gcam2) $(TESTS) AutoTest(test))
...
# Macro to locate IMP Scratch for tests below.
LOCALIMP=IMP_SCRATCH=`pwd` ; export IMP_SCRATCH ;
# Macro to load a task using ditsloadcmd
LOADTASK=$(LOCALIMP) $$DITS_DEV/ditsloadcmd -mayload
# Macro to send a command. -n option fixes the name rather then PID based.
DITSCMD=$(LOCALIMP) $$DITS_DEV/ditscmd -n DITSCMD
# Macro to cleanup IMP, if needed (if there are tasks running)
IMPCLEAN=$(LOCALIMP) if [ "`$$DITS_DEV/tasks`" != "" ]; then ($$IMP_DEV/cleanup >/dev/null 2>&1 ; sleep
1) ; fi
test : test1.out test2.out

test1.out : Lib(gcam2) gcam2test1 CameraServer test1.expected
      - $(RM) test1.out test1.out.tmp
       $(IMPCLEAN)
      ./CameraServer &
      $(LOADTASK) -mayload ./gcam2test1 > test1.out.tmp
      $(DITSCMD) GCAM2TEST1 INITIALISE >> test1.out.tmp
      $(DITSCMD) GCAM2TEST1 EXPOSE 3 >>test1.out.tmp
      $(DITSCMD) GCAM2TEST1 READOUT >>test1.out.tmp
      # Not putting these two in test1.out.tmp - floating point issues.
      $(DITSCMD) GCAM2TEST1 CENTROID 100:100:20 10 2
      $(DITSCMD) GCAM2TEST1 CENTRECT 10:10:40:50 10 2
      $(DITSCMD) -x GCAM2TEST1 SHUTDOWN_CAMERA >>test1.out.tmp
      diff test1.out.tmp test1.expected
```

```
mv test1.out.tmp test1.out
@ echo "gcam2test1 test complete."
```

9 Ensuring translations of error codes.

DRAMA uses integer error in a well defined format. Each task/library has its own "facility number" which defines the set of error codes it can provide. The template AAO DRAMA tasks generated when you create a new AAO Sub-system creates a facility number and generates a template error code definition file (A <task>_err.msg file). The full details are documented in the messgen document.

9.1 Translation by tasks

The template AAO DRAMA task created when you create a new ACMM sub-system will correctly process the "<task>_err.msg" file to create the "<task>_err_msgt.h" and "<task>_err.h" files. The former associates strings with each error code, the later just defines macros for each error code. Again all the details are in the "messgen" documentation.

Within your code, you include the "<task>_err_msgt.h" once per program and invoke "`MessPutFacility()`" to enable translation of the codes.

Within your code, you include "<task>"_err.h in each file and use it to get error code numbers (e.g. to set in exceptions)

It is important that a DRAMA task also properly load the error code translations for all libraries it uses and all other tasks it runs. For each such library/task, you should include the "<task>_err_msgt.h" file and invoke "MessPutFacility()". This will ensure your task can correctly translate all the relevant error codes it might receive.

9.2 Translation by ditscmd

From DITS version 3.111 and DUL version 3.60, "`ditscmd`" can translate error codes which are NOT built into it.

Previously it could only translate built in error codes, which meant only the basic DRAMA error codes. For example, you might see:

```
DITSCMD_13a7:exit status:%NONAME-E-NOMSG, Message Number 0C328022
```

Which doesn't help much.

To enable translation of these codes, you need to have the error code facility file listed in the value of the `DRAMA_FACILITIES` environment variable. You can generate the value required by knowing the systems you are running and/or by using the "`messana`" command to analyse untranslated codes. For example,

```
>>messana 0x0C328022
Message c328022(204636194) - Facility:1074(NONAME), Number:4, Severity:2
Uknown facility
```

(Note the 0x prefixed to the above value to ensure messana knows it is a hex value).

From this, we now know the error facility number is "1074".

From there, use `acmmPrintAAOErrorCodes` to find out where this comes from. E.g.

```
>> acmmPrintAAOErrorCodes | grep 1074
Facility number 1074, Module TaipanCANBusController, 08:20:05 04-Apr-2016, user mvv@aaomc66mvuo.aao.gov.au
```

So this tell you it most likely comes from the `TaipanCANBusController` sub-system. So that the error code facility I am missing. You need to find the "`_msg_t.h`" file associated with the sub-system and add it to the value of the `DRAMA_FACILITIES` environment variable. Each instrument's login account files should ensure this is set correctly. For example, in TAIPAN, the Control task (TICT) releases the file `drama_facilities.sh` which sets this environment variable. This is sourced by the ".drama.sh" file of accounts which might run Taipan. Here is its contents as of the date of writing this:

```
export DRAMA_FACILITIES="\
TAIPANJSON_DIR:taipanJSON_err_msgt.h \
TAIPANPOSCOORD_DIR:tpc_err_msgt.h \
DRAMAUNIXSTART_DIR:dramaunixstart_err_msgt.h \
TAIPANTCS_DIR:taipantcs_err_msgt.h \
```

```
TAIPANAANDG_DIR:taipanAAndG_err_msgt.h \
TAIPANDRAMAPOSITIONER_DIR:taipandramapositioner_err_msgt.h \
SB_DRAMAMOTIONCONTROLLER_DIR:sb_dramamotioncontroller_err_msgt.h \
TAIPANMETMODEL_DIR:taipanMetModel_err_msgt.h \
GCAM2_DIR:gcam2_err_msgt.h \
GCAM_DIR:gcam_err_msgt.h \
DSERV_DIR:imgrtd_err_msgt.h \
DSERV_DIR:dserv_err_msgt.h \
DRT_DIR:drt_err_msgt.h \
DRTF_DIR:drtf_err_msgt.h \
/instsoft/extern/include/taipan/taipanmetdramatask_err_msgt.h \
/instsoft/extern/include/taipan/taipanmetrology_err_msgt.h \
/instsoft/extern/include/taipan/taipancanbuscontroller_err_msgt.h \
"
```

The actual definition of the value is a space separated list of files (just one line). But the DRAMA environment variable translation approach can be used - the "ENVVARNAME:filename" format. This defers translation until the underlying code looks up the file.

In this example, most files are found that way (they are released in the DRAMA release of the sub-system). A few are not since they are released differently, and the full filename can be specified.

Having done this, the above example looks like this

> *DITSCMD_20bd:exit status:%TAIPANCANBUSCONTROLLER-E-INIT_AMP_FAILED, Failed to initialise Copley amplifier*

Which is much clearer.

10 DRAMA - releasing subsystems (AAT style)

This page is describing the process for working on and releasing DRAMA sub-systems - particularly the approach traditionally used at the AAT, but adapted for the move from ACMM to Git as the repository.

First please note that much (but certainly not all) AAT instrument software is built on "aatlxy". This is a relatively old machine. At about 0UT time each day, its /instsoft disk is synced to aalxx, so that either of these two machines may be used.

10.1 "git" on aatlxy/aatlxx

The "git" software was not available in a package for aatlxy/aatlxx (too old). But it has been built by hand for these machines. Please add this directory to your path:

```
/instsoft/extern/git-x86/bin
```

10.2 "git" conflict with DRAMA.

DRAMA uses the GIT_DIR environment variable at run and build time. This conflict with the "git" repository software which also uses that name.

If you are using bash/zsh as your shell, please add the following to your "~/.drama.sh" file

```
git()
{
     env -u GIT_DIR git "$@"
}
```

If using tcsh, then add an equivalent alias to your "~/.drama.csh" file.

This seem to do the trick in most cases, but any script which uses git and might run when DRAMA is enabled will also need that “env -u GIT_DIR” prefixed to each "git" command.

10.3 First time change to "dmakefile"

This change only needs to be make the first time a DRAMA repo modified after this approach was introduced (in 2022).

Each DRAMA repo (other then some which have had complex modifications since being moved to GitLab, contain a file named "dmakefile". The older ones will contain these two lines:

```
ACMM_RELEASE=11_44
RELEASE=r$(ACMM_RELEASE)
```

This was the path to getting the release number updated automatically by ACMM. Replace these two lines by"

```
XCOMM If your makefile fails at the following non-comment line below, and still has
the line SetGitRepoRelease, then, either update DRAMA OR
XCOMM   Add "#define SetGitRepoRelease() GIT_REPO_RELEASE:=$(shell env -u GIT_DIR
git describe --dirty)" to $DRAMA_LOCAL/drama_local.cf
SetGitRepoRelease()   /* Required to ensure GIT_REPO_RELEASE is set efficiently */
RELEASE=r$(GIT_REPO_RELEASE)
```

Now the release number will be based on the git tags, but it does require a modification to $DRAMA_LOCAL/drama_local.cf (or a later version of DRAMA which includes the definition internally)

I have already set up the macro GIT_REPO_RELEASE in $DRAMA_LOCAL/drama_local.cf for aatlxy/aatlxx. Some other machines about may not have it. The comment in the change explains what is needed

10.4 Git approach

Enable drama (e.g. ~drama/dramastart -v <current_version_number>)

Clone to working machine (E.g. aatlxx/aatlxy. Of these two, prefer aatlxy since that is the source of the sync, but if after 11am DST and you want to make a change for tonight, then you need to be on aatlxx)

Modify dmakefile as above - one time only.

dmkmf, code, make, make release, test.

At this point you will be releasing into a directory tree which indicates it is based on a dirty repo, e.g

```
TDFCT_DIR=/instsoft/drama/local/tdfct/r12.57-dirty
```

When happy, commit and tag. In the above case, the new tag would be 12.58 (e.g. no “r”, that comes from the dmakefile)

Once you have tagged, again do

make clean, make, make release

(Since most of the executables have the version number built into the runtime - you need a clean build after you have tagged - there is probably a quicker way (only one object to be rebuilt) but I haven’t put the effort into working that out yet).

The release directory should now be based on the new tag, e.g.

```
TDFCT_DIR=/instsoft/drama/local/tdfct/r12.58
```

And then copy $DRAMA/<current_version_number>.ver to a new file, and change the entry for each subsystem

And don't forget to push your repo (most likely in a new branch) and also to push your tag.

10.5 To help in Developement

```
make dramadirs
```

Outputs the environment variables. So when I am working on something, I normally do

```
eval `make dramadirs`
```

to test against the new version without having to modify the version file in ~drama and restart DRAMA. There is also “make dramadirsl”, which sets the environment variables so your local directory.

There are also versions "make dramadirs2" and "make dramadirs2l", which output the appropriate commands for csh/tcsh.

And “make releasenum” will give you what goes in the version file (used for automatic building of the version files).