DRAMA – A brief overview

Research Data and Software
Australian Astronomical Optics,
Macquarie University.

September 2019

DRAMA – A brief overview

1 Introduction.....3
2 Astronomical instruments.....4
3 A DRAMA task from the outside.....7
3.1 Actions.....7
3.2 Kicks.....7
3.3 Triggers.....8
3.4 Parameters.....8
3.5 Parameter monitoring.....9
3.6 Overall.....10
3.7 SDS and Data structures in DRAMA.....10
3.8 Real-time aspects.....12
3.9 DRAMA tasks working together.....13
4 DRAMA tasks from the inside.....14
4.1 Programming languages.....14
4.2 Multiple concurrent actions, and multi-threading.....15
4.3 The structure of a DRAMA task.....16
4.4 The structure of a DRAMA action.....16
4.5 DRAMA messages.....20
4.6 Other DRAMA facilities.....21
5 DRAMA 2.....22
5.1 DRAMA2 actions.....22
6 Summary.....26
7 DRAMA Glossary.....26

1 Introduction

DRAMA is a software environment that simplifies the writing of multi-tasking data acquisition systems. These systems may involve a number of different types of computers and operating systems. It was originally written at AAO[1] in

[1] Originally the Anglo-Australian Observatory, later the Australian Astronomical Observatory, now Australian Astronomical Optics.

the 1990s and has been used extensively at AAO and other observatories. From 2015,, it has been re-invigorated through the development of DRAMA2, which builds on the original DRAMA system but which adds support for multi-threading. This makes it much easier to code single tasks that need to support multiple concurrent operations. DRAMA has always supported such tasks, but multi-threading support makes the coding much simpler. Additionally, DRAMA2 takes advantage of C++14 features to provide a much more modern API for DRAMA.

This document attempts to introduce the concepts behind DRAMA. It is not a DRAMA manual, and it contains almost no examples of actual code. It is intended simply as an introduction to what DRAMA has to offer. It should also serve as useful background for anyone who is about to dive in to the detailed DRAMA API manuals in order to write a DRAMA task or to understand how an existing task works.

2 Astronomical instruments

Although a general-purpose system such as DRAMA can be used in many different ways, if you want a feel for its design and philosophy, it helps to look at the sort of problem it was originally designed to solve.

Look at what is being used during a typical night when an optical telescope such as the AAT[2] is being used.

Light comes through the telescope itself, through a spectrograph, onto a detector, and the resulting data is[3] displayed, recorded, and processed, at least sufficiently to gauge the quality of the data being obtained. There are a number of separate instrumental systems involved here (telescope, spectrograph, detector) which need control software, and a number of jobs that are being performed purely in software (data recording, data display and data processing). Some software also has to be in overall control of all this, coordinating all the other parts, and there must be a way (inevitably GUI-based) for humans (observer, telescope operator, etc) to interact with the system. Taking a very big-picture view of all this, there are a number of separate, reasonably big ‘blocks’ to the system, and these are shown in Figure 1.

---

[2] The 3.9m Anglo-Australian Telescope, at Siding Springs in NSW, Australia.

[3] OK, pedants. The datums are…

Note that what this block diagram shows is intended to be a set of 'functional blocks'. Some are purely software, but some, like the telescope and spectrograph, will be physical instruments that have software components.

It's worth noticing that any telescope may operate with a number of different instrumental combinations, but that the same structure may be maintained, and many of these 'blocks' may be reused in different combinations. The telescope will probably always be there, and while there may be different spectrographs in use, it may be that the same detector sub-system is used for different spectrographs. Data recording, display and processing will be similar for most instruments, and may be identical for many, especially if they are written in a sufficiently general way.

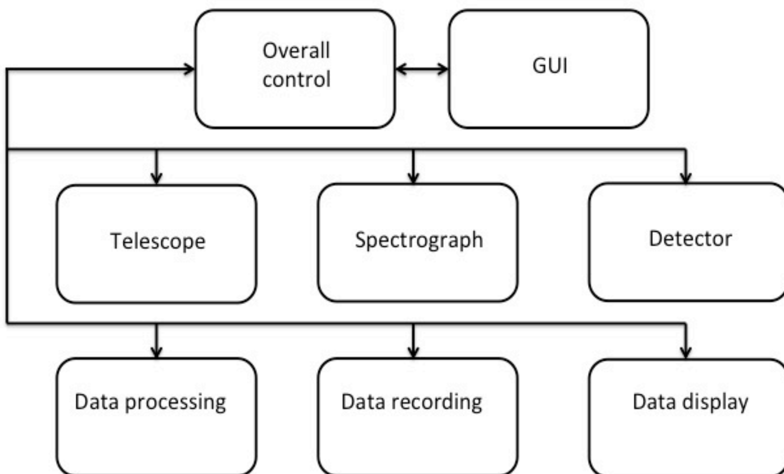

Figure 1 Overall observing structure

So it makes sense for the structure of the software system to match that of the blocks shown here. If each of these functional blocks has its own associated software block, and if these software blocks exist independently (as opposed to being just a part of some large monolithic single program) then new instrument combinations can be created by, for example, introducing a new spectrograph, and the corresponding overall software control system merely requires a new spectrograph software block.

All of this is fairly obvious, and is true of almost any sufficiently complex software system. DRAMA predated the general popularity of C++, but most of

the considerations that went into DRAMA can be seen nowadays in object-oriented design: modularity, encapsulation, well-defined interfaces. The difference is that DRAMA implements all this at a high level, that of individual tasks, while C++ does it at a much finer level[4].

In a DRAMA system, we would have a separate software task for each of these functional blocks. 'Task' in this context generally means an independent, executable program, capable of being linked separately and run separately. (Usually, each DRAMA task runs as part of a system, as shown in Figure 1, but it can be run separately and can be tested separately – the testing is an important aspect.) This is the meaning of 'task' as used by most operating systems[5].

It is, of course, possible for any of these tasks to be implemented using multiple tasks. In practice the AAT telescope control task comprises eleven separate DRAMA tasks, but only one of these, the 'main' TCS task, is normally visible to the rest of the system and all communication goes through it[6].

So what DRAMA provides is a model of how such a task looks to the outside world, and the libraries that can be used to implement such a task.

There are two parts to this. There is how a DRAMA task appears externally, and there is what it looks like internally. The external model is really quite simple, but is very flexible, particularly in the way data structures are used to pass information. The internal details can get quite complicated, and part of the aim of the DRAMA2 work was to simplify the coding of a typical DRAMA task, particularly in terms of how it handles concurrent operations. The following sections describe first how a DRAMA task looks from the outside, and then looks at what goes on inside to produce such a task.

As with all systems, there is some nomenclature to get used to. Words like 'action' and 'kick' and 'trigger' and even 'parameter' have particular meanings in a DRAMA context.

[4] The two can work together, of course. A DRAMA task can be built out of C++ objects, and there will often be a single C++ object that implements the task itself.

[5] But note that on systems such as older versions of VxWorks, a task may not have its own separate address space, and one can argue about whether it is really a task or a thread.

[6] For diagnostic purposes, however, it is possible to send messages – for example, enquiring about detailed task status – to individual component tasks, either from the command line or some other engineering interface.

3 A DRAMA task from the outside

A DRAMA task is a separate task that has a name, which it uses to register with the DRAMA system, and such a task can be sent messages through the DRAMA message system. It can send messages back. It is expected to remain continually responsive to messages sent to it.

3.1 Actions

A DRAMA task implements a set of named 'actions'. An action can be invoked by sending an 'obey' message to the task, specifying the name of the action and, optionally, parameters[7]. For example, a telescope control task might have an action called 'SLEW_TO' which might take parameters describing the position of the target object. A detector control task might have an action called 'EXPOSE' that takes a parameter giving the required exposure time.

An action does not have to complete immediately, although many do. When it does complete, a message will be sent back to the requesting task. An action that has been invoked (through an 'obey' message) but has not yet completed is described as being 'active'.

A task may have any number of actions active at the same time. It is possible to have multiple concurrent instances of the same action, but this is unusual and needs more careful handling.

3.2 Kicks

It is possible to send a message to an active action in a task in order to influence the course of that action. An obvious example would be the case where, for some reason, a task wants to cancel an action it has invoked in another task. If it gets cloudy, one might want to cancel a detector control task's 'EXPOSE' action. Or one might want to extend the exposure time, which modifies the course of the action but does not cancel it. In the AAT control system, one of the component tasks calculates a new demand position for the telescope every twentieth of a second, which it sends in a message to another component task whose job it is to apply this demand position. Such a message

[7] Strictly, what is sent is a single 'argument structure', but this can be arbitrarily complex and can contain any number of sub-structures that can be treated as individual parameters. We tend to describe actions as having a number of named parameters, but they're actually named sub-structures of the one argument structure.

sent to a task in the context of an active action is termed a 'kick' message; it 'kicks' the action in a certain way.

Just like an 'obey' message, a kick message can have an arbitrarily complex data structure associated with it, and this can be used to supply parameter values for the kick.

The requirement that a DRAMA task remain always responsive to incoming messages comes partly from the need to be able to send 'kick' messages at any time and have the task act on them without delay.

3.3 Triggers

An action can send a message back to its invoking task at any time. This might, for example, be a progress report. Such a message is called a 'trigger' message[8]. Trigger messages can be used for any purpose, and – like most DRAMA messages – can include an arbitrarily complex data structure that can be used to supply associated information.

3.4 Parameters

A DRAMA task can also have any number of named parameters. Each of these is (you can probably guess this by now) an arbitrarily complex data structure consisting of named components. One DRAMA task can send a 'get' message to another DRAMA task specifying the name of such a parameter, and it will be sent back a reply message that contains a copy of that parameter.

It is also possible for one DRAMA task to send a 'set' message to another DRAMA task in order to set the value of a named parameter. In practice this feature is not used very often; a 'set' effectively changes the value of a parameter behind the back of the owning task, and it's usually better to implement an action called something like 'SET_name' which takes the new parameter value as its argument structure. If you do that, then the task-specific code in the owning task knows that the parameter has been changed, and usually that is something you do want to know.

But task parameters really come into their own when you monitor them.

[8] 'Kick' is a fairly obvious name for a message that influences the course of an action. 'Trigger' for a reporting message, somewhat less so. All systems have some historical quirks.

3.5 Parameter monitoring

A DRAMA task can 'monitor' any number of parameters in other DRAMA tasks. Whenever the value of a monitored parameter changes, each task that is monitoring that parameter receives a DRAMA message that has a copy of the new value of the parameter.

This is useful in a number of circumstances, but is particularly useful for GUIs. A very convenient way to write a GUI for a DRAMA task is to have the GUI monitor a number of parameters in one or more other tasks. Usually, such parameters exist purely in order to be monitored. When a task makes a change that needs to be reflected in a GUI, it simply changes the value of one of these parameters. The GUI task gets a message with the new parameter value, and it can modify its display to reflect it. This is much, much, simpler than requiring the monitored task to send explicitly send a message to a GUI task whenever it makes a change it wants to see reflected in the GUI. No housekeeping is required in the monitored task , other than setting the parameter value; it does not have to be aware of the identity of the GUI task, or even if it exists at all – if the GUI task is not running, everything continues to work.

A number of instrument control systems, such as EPICS, operate as 'distributed databases', where the overall system is seen mainly as a collection of database values all of which are linked, so that as one changes, this change is picked up by other parts of the system. In such systems, the values involved can be very small-scale quantities, such as the temperature of a detector or a particular control voltage. DRAMA parameter monitoring can also be used to set up this sort of interaction, and its design was originally influenced by such systems. DRAMA is usually used as a 'command-based' system, where tasks send explicit commands (usually at quite a high level, like 'task an exposure') to other tasks. The display of values in a GUI, however, is something one wants to be triggered by changes to individual small items such as temperatures or voltages, and parameter monitoring works very well in such cases[9]. It also means that DRAMA has the flexibility to be used in systems that make use of database concepts, or at least in some sort of hybrid manner.

---

[9] As an example, in the AAT TCS, the raw encoder readings, both incremental and absolute, for both the telescope axes are read by a separate 'Encoder' task which writes them to a parameter which is monitored by an engineering section of the control GUI task, which displays them both as binary patterns and as RA and Dec positions.

3.6 Overall

And that is pretty much all there is to a DRAMA task, as far as its interaction with the rest of the system goes. It has a name, it supports a number of named actions, and it maintains a number of named parameters. Its actions can be kicked, and its parameters can be monitored. That is a very simple interface. (It isn't unlike the interface of a C++ class, which has a number of methods that can be called, and a number of instance variables whose values can be accessed – so long as they are made public. DRAMA tasks can even inherit from other DRAMA tasks, in a way, but that comes later…)

Note though, that a lot of the power of this simple interface comes from the use of structured data for message arguments and for parameters. That needs a little more detail, before we look at the internals of a DRAMA task.

3.7 SDS and Data structures in DRAMA.

SDS – the Self-defining Data System – is a library that provides a lot of the flexibility in DRAMA. It creates hierarchical structures in memory. Each structure can contain any number of named items, which can be other structures or can be primitive data (floating point values, integer values, or characters, as either single quantities or as multi-dimensional arrays). There are routines to create such structures, to locate named items within them, to get their type and dimensions, and to search through structures to see what they contain. ('Self-defining' refers to the fact that all the details about the structure contents – although not their meaning – are contained within the structure itself.) The structures are mutable when created, although they can be 'exported'[10] in a fixed form into either files or into contiguous regions of memory. The various enquiry and data access routines work on either the mutable or fixed forms.

The size and complexity of SDS structures are limited only by machine resources. It is quite common to store multi-megabyte images or even data cubes in SDS structures. The size of DRAMA messages is also essentially unlimited[11].

---

[10] 'Serialised' might be a more modern term.

[11] When a task registers or sets up a communication path with another task it has to specify the maximum amount of message size to be allocated for the purpose. However, the size that it can specify is limited only by machine memory constraints.

When DRAMA sends an ‘obey’ or other message, that message can include a fixed SDS structure as an argument.

This means that almost anything, including large images, can be included in DRAMA messages[12].

As an example, the section on ‘kick’ messages mentioned such a message being sent every twentieth of a second to one of the TCS tasks to give it the new telescope demand position. Figure 2 shows the contents of the argument structure associated with this ‘kick’ message. The details are unimportant, but it can be seen that there are a lot of values specified in this message, each named, and all look as if they might be something to do with telescope position (TAI is the time, for example). This is not a large message – but it is sent twenty times a second.

```
ArgStructure              Struct
   TAI                      Double 57308.22076
   A                        Double 0.7387388122
   B                        Double -0.6959722159
   AV                       Double 7.291577214e-05
   BV                       Double -4.401439213e-09
   NEW_POSN                 Int    0
   TRACKING_OK              Int    1
   AZ                       Double 4.257689448
   EL                       Double 0.9576399682
   AZV                      Double 1.027695049e-06
   ELV                      Double -5.597003429e-05
   DOME_DIRECT              Int    0
   STZERO                   Double 4.354585811
   MJDZERO                  Double 57308.21938
   XGD                      Double 4.2095517e-09
   YGD                      Double -6.914826319e-11
   AUTOGUIDE_REQ            Int    0
   GUIDING_ZERO             Int    0
   APPHA                    Double 0.7388443341
   APPDEC                   Double -0.6952868241
   NEW_TARGET               Int    0
```

Figure 2 AAT TCS Kick argument structure

[12] For truly huge amounts of data, where the overheads of copying into DRAMA messages would be excessive, DRAMA also provides a very efficient alternative ‘bulk data’ sub-system that uses shared memory to eliminate most data copying overheads.

SDS can be used as a hierarchical file format, and you will find files with a .sds extension in many DRAMA-based systems, but its real power comes from the ability to create, manipulate, and transmit complex hierarchical, searchable, data structures in memory.

DRAMA makes use of SDS almost anywhere it needs to transmit data values. Rather than send a fixed set of parameters as arguments to an action, it sends an SDS structure. The receiving action can see what it has been sent, searching the structure for named items that it expects as parameters. When it finds a parameter, it can see what form it takes: it might be a scalar, it might be a multi-dimensional array, and the task may use the parameter in different ways depending on just how it has been specified. If it is not present, it can use a default value instead. When a task responds to a command, it sends back an SDS structure as part of its response. This can be anything from a single byte set to some status code, or it can be a comprehensive set of diagnostic information.

Almost anything can be put into an SDS structure, and almost any SDS structure can be sent as part of a DRAMA message, or used as a DRAMA task parameter.

3.8 Real-time aspects

DRAMA tends to be described as a 'soft' real-time system. Usually this is taken to mean that it can be expected to provide a reasonably reliable real-time response, but you should not expect it to be able to handle requirements such as 'this message must be handled within n microseconds of its being generated'. However, this is selling it somewhat short.

It is true that DRAMA usually runs on systems like UNIX or OS X, which are certainly not normally 'hard' real-time systems. However, it also supports VxWorks, which is regarded as a proper real-time system, and could easily be ported to similar systems. AAO systems that require a proper real-time response are implemented using DRAMA tasks running on VxWorks. Hard real-time is not easy, but DRAMA does have facilities that support it. Even under ordinary non-real-time UNIX, you can get a real-time response by writing purpose-built drivers. In particular, it is possible for a real-time thread, or an interrupt service routine, to send a DRAMA message to a DRAMA task. So a high-priority thread or interrupt routine can be used to get the required real-time response, and can work in conjunction with a DRAMA task. Such a thread cannot, of course, wait for a DRAMA message itself, but it can communicate directly with a task that does.

One of the requirements of the original DRAMA design was that it should be capable of handling all aspects of data acquisition. At AAO we use OS X to write and test DRAMA tasks that will eventually be deployed on VxWorks. Having the same DRAMA API available on a range of platforms, including those that operate at the hard real-time end of the spectrum, makes software development much simpler. It becomes much easier to develop even those parts of the system that will run on systems that can be tricky to work with; testing hard real-time programs is never easy, but it helps if most of the less time-critical code can be developed and tested on other systems.

3.9 DRAMA tasks working together

The diagram shown in Figure 1 shows a typical DRAMA task configuration. An overall control task communicates with a number of individual tasks, each with a particular area of responsibility. In this case, the task layout follows the overall system structure quite closely.

The detector control task, for example, is responsible for all control of the detector. At least, that is how it looks to the rest of the system. In practice, it probably is really just an interface to the detector control hardware, and may even be just a thin layer between the other DRAMA tasks and a separate non-DRAMA layer of specialised control software geared to detector hardware.

The important point is that it presents to the outside world a detector system that responds to commands such as 'EXPOSE', 'READOUT' and the like, and which provides a set of status commands and/or parameters that show the state of the detector. It might have a command like 'RECORD' that takes a filename as an argument, or it might just have a parameter that is a copy of the last-read image. There are a number of ways such a task could be specified.

In this example, the task layout is relatively static. Most of these tasks can be loaded as the system starts up and stay running all the time. And these tasks are relatively large and complex, handling all aspects of whatever it is they control.

But it is equally possible to have a system that uses a set of small, transient, tasks that come into existence briefly, do one thing, and exit. A good example is the general purpose DRAMA utility program 'ditscmd'. This is a command line utility that takes as command line arguments a target task name, and some additional arguments. It runs, connects to the target task, sends the required message, waits for a response, and exits. So, the command:

```
ditscmd FRED GET_STATUS
```

will invoke the GET_STATUS action in the task that has registered in the DRAMA system as 'FRED'. It will wait for the reply, which presumably will be accompanied by an SDS structure that contains details about the status of the task, and will display the contents of that structure. This program can also be used to display the contents of a task parameter:

```
ditscmd –gv FRED STATUS_PARAM
```

will send a 'parameter get' command to FRED, requesting the value of the 'STATUS_PARAM' parameter, and will display the result. (The –g indicates a 'get' command, and the v indicates 'verbose', which makes ditscmd list the whole hierarchical structure of the parameter, including all the item names and types.)

It is possible to use DRAMA as the basis of a command line-based data reduction system, each command starting up a new task that performed some specific data reduction task on files and with parameters named in its arguments.

In the end DRAMA is a tool, and because it is a very flexible tool, it can be used in a number of different ways – not always those envisaged by its creators.

4 DRAMA tasks from the inside

This section looks briefly at how a DRAMA task is structured internally to get the effects already described.

4.1 Programming languages

First, a note about languages. The original DRAMA API used C, and DRAMA was implemented entirely using C. To make it easier to write GUIs for DRAMA, a Tcl/Tk interface was written, and it is possible – and was common – to write DRAMA tasks in Tcl, using Tk to provide a GUI.

It is possible to write a DRAMA task that combines both C and Tcl/Tk in order to create a task that implements most of its actions in C but also provides its own Tcl/Tk-based GUI, all within the same task. An alternative is to write a DRAMA task entirely in C (or, more usually nowadays, C++) and have it set parameters that a separate Tcl/Tk DRAMA task can monitor., as described earlier. This arrangement allows the C-based DRAMA task to run without a GUI if required, but allows the GUI task to be loaded when needed. It also tends to lead to a simpler, more modular design for each task.

There is also a Java interface for DRAMA, and a DRAMA task may equally well be written using Java.

A Python interface for DRAMA is has recently been implemented. It is not yet as complete as the older Tcl/Tk interface or JAVA interfaces, in that you can't yet implement all features using Python. But Python is now being used for scripting and GUI can be implemented using many python packages.

Another alternative for tasks that need to provide a GUI is to use Qt. Since Qt is a C++ library, it can be used fairly straightforwardly with DRAMA.

4.2 Multiple concurrent actions, and multi-threading

Most of the issues with the internals of DRAMA tasks arise from the need to support multiple concurrent actions. For example, a spectrograph might have, amongst other things, a grating that can be set to a range of different angles, a filter wheel that can be set to any one of a number of different positions, and a slit that can be opened to any of a range of widths. One might design the spectrograph control task to have a SLIT action that took a width parameter, a GRATING action that took an angle as a parameter, and a FILTER action that took either a filter name or a number as a parameter[13].

The original DRAMA design came from an era when multi-threading was unusual and was not provided by many operating systems[14]. So DRAMA did not use multi-threading to implement concurrent actions. Instead, it used what amounted to a form of 'cooperative multi-tasking' within a task.

Because most existing DRAMA tasks use this original DRAMA design, it will be described first. Note, however, that the more recent DRAMA2 API implements multiple concurrent actions through the use of multiple threads. This will be described later. DRAMA2 also provides a much more modern, C++14-based API, and this means that the code for a DRAMA2 task can look rather different to that for a DRAMA task. However, with the exception of the multi-threading aspects, most of the concepts are the same as those in the original DRAMA.

---

[13] Because the parameter passed when the FILTER action is invoked is an SDS structure, the code can look to see if the parameter has been specified an integer or a character string, and interpret it accordingly.

[14] To put things into context, nor was the socket() call provided on all machines!

4.3 The structure of a DRAMA task

It is convenient to think of a DRAMA task as having a 'fixed part' and a 'task-specific part'. The 'fixed part' is the same for all DRAMA tasks, and provides the basic flow of control for the task.

In essence, the fixed part waits for a message from another task, and handles it. There are a number of internal DRAMA messages that are handled entirely within the fixed part. Messages connected with the actions defined by the task are handled by callbacks from the fixed part to routines in the task-specific part that have been registered as the 'action handlers' for the various actions. This description applies to both the original DRAMA design and to DRAMA2, but the way action handlers are expected to act is significantly different between the two APIs. What follows now applies only to the original DRAMA design.

4.4 The structure of a DRAMA action

Remember that the DRAMA concept requires that an action can be interrupted[15] by a 'kick' message. For a DRAMA task to remain responsive, this means that any action handler invoked in response to a message must return immediately[16], because the fixed part cannot read that message until it does so.

Some actions will be able to complete entirely in a sufficiently short time. For example, an action that merely returned a status report could do this easily. An action that slewed a telescope to a new position, or which handled a detector exposure of many seconds, clearly cannot return 'immediately'. DRAMA2 can handle these cases using multi-threading, but in the original DRAMA system, such actions have to be 'staged' – split into separate stages, each of which can be handled very quickly.

---

[15] This is not 'interrupted' in the sense of a hardware interrupt handled by an interrupt service routine; it just means that once an action has started, it must still be possible to send a message to it that will be acted upon immediately. The most useful example to think about is the case where the message is a request that the action be cancelled.
[16] Just what 'immediately' means depends on the context; if an action handler takes a tenth of a second before it returns, the task will be unresponsive for that tenth of a second. For some tasks that would be acceptable, for others this would be too long. For most 'soft' real-time systems, something like a tenth of a second is usually acceptable, and is a good rule of thumb.

Take the case of an 'EXPOSE' action sent to a detector control task. This will be invoked by a message that requests that the EXPOSE action be started, and which supplies an SDS structure containing the details of the exposure – these could be quite complicated, but at the very least will probably include an exposure time.

When the task in question starts up, its main() routine is invoked. This performs some initial setup for the task, and then calls the standard DRAMA routine that runs the DRAMA main loop (the loop that reads new messages, calls the routines that handle them, and then reads a new message). As part of the setup, the DRAMA parameters for the task are usually created, and action handlers are registered for all the various actions supported by the task.

So when the message requesting the EXPOSE action is received, the DRAMA fixed part knows which routine in the task-specific part was registered as the action handler for that action, and it calls it. This action handler routine can tell that it has been called as the result of an 'obey' message being received, and it can access the SDS structure included with the message. It can extract the parameters such as the exposure time from that structure, and can initiate the exposure, presumably by communicating in some way with the detector hardware. It can probably start the exposure[17], but it certainly can't make the whole task wait until the exposure completes – otherwise there would be no way in which a request to cancel the exposure would get through. So it does what it can in an acceptable time, then returns to the DRAMA fixed part.

When we say it 'returns to the DRAMA fixed part' all we mean is that the action handler routine returns to its caller. Since it was called from the fixed part, the fixed part now has control again. But before the routine returns, it calls a standard DRAMA routine that lets it tell the fixed part what it wants to happen next – it makes a 'request' of the fixed part. It can request one of a number of things:

- It can request that its next stage be invoked immediately. In this case, it is essentially saying 'I can carry on right now, and would like to, but I'm returning to give you a chance to see if there's anything come in that needs to be handled.' (And one thing that could have come in might be a request that this action be cancelled, of course.)

---

[17] Just how easy this is, of course, depends on the detector hardware. It may require a complex interchange of (non-DRAMA) messages with the hardware before the exposure starts, in which case even this initial sequence will have to be 'staged'. This is a more advanced topic.

- It can request that its next stage be invoked after a specified time. This can be used to allow the action to run in a 'polling' mode; constantly rescheduling on a regular basis to see how some external system is going[18].
- It can request that its next stage be invoked when a new message arrives in the context of this action. That is, there is nothing more it can do at the moment, but it is expecting something to happen which will cause it to be sent a message. (Often this is a message from another task with which this action is communicating – it might have requested a connection to another task and is waiting for that to be set up. In some cases, it can be more complex – it may be waiting for a hardware interrupt whose service routine will generate a DRAMA message, for example. Or it may just be idling and is waiting to be told what to do next. DRAMA supports a lot of possibilities.)
- It can request that this action end now. This usually means the action has either completed successfully or has run into an unrecoverable error.
- It can request that not only should the action complete now, but that the whole task should exit.

In all these cases (well, except the case where the task exits) the fixed part now takes control again and goes back into its main 'read a message, handle it, read a new message' loop. This scheme, with all actions split up into small discrete stages, allows any number of actions to be active concurrently, and the fixed part will reschedule them, stage by stage, as required.

At start-up, action handlers are registered for all actions, and kick handlers are registered separately for those actions that need them. Generally, any action that is going to do anything other than complete immediately will need to register a kick handler.

It is possible for an action handler to specify a new routine to serve as the action handler for the current action. This allows a style of coding where, each time an action stages, it specifies a new routine to handle the next stage of the action. There are some actions that move through a clear sequence of separate stages, and where this is the case, this style can be rather easier to follow. The alternative is to have the same action handler set for the whole of the action, in which case the code in that action handler needs to check just why it has

---

[18] For those who like to know how such things are arranged, in this case the fixed part will schedule a timer that will send it a message once the specified time has expired. So then all it has to do is re-enter its normal message loop.

been invoked each time it is called (is this a new invocation of the action, or is it a restaging, or has a message been received, etc?). It is because the action handler can be changed dynamically in this way that it is convenient to be able to specify the kick handler separately, as kicks to the action may come through at any time. Each time an action completes, the action handler reverts to that specified at start-up, which is generally what is wanted.

This scheme is essentially a form of 'cooperative multi-tasking[19]' within a single task. This is the form of multi-tasking used by early versions of both Windows and MacOS, and is the only form available in the absence of support for pre-emptive multi-tasking. So at one time this was quite a familiar way of programming. (And now that this is no longer the case, we have DRAMA2, which essentially takes advantage of the pre-emptive way in which multiple threads work within tasks.)

In a multi-tasking operating system this is very much a second-rate way of multi-tasking, mainly because it allows a rogue – or just plain badly written – task to monopolise the whole system. This objection is relatively unimportant in the context of a DRAMA task, where usually the same programmer is responsible for the whole task, and can hardly complain about the poor standard of coding. What it does have is one significant advantage, and one significant disadvantage:

- The advantage is that forcing the code for an action to return to the DRAMA fixed part on a regular basis not only allows concurrent actions to run, it also allows actions to be cancelled easily. There is never a question of how one cancels a thread that is blocking on some operation, since threads aren't allowed to do that sort of thing. Also note that what actually happens is not that the action is cancelled from outside, which can often be problematic, but that the action is sent a request that it cancel itself, which it can usually do cleanly.
- The disadvantage is that while some actions can be easily coded as a set of short stages that can easily return to the fixed part as required, some actions are very awkward to code in this way. (And it is no longer a programming style that many programmers are familiar with.) Any action that involves a long sequence of exchanges of messages with either another task or a piece of hardware does not fit naturally into this style. It can be done, but it is awkward, frustrating, and for these reasons, error prone. It also encourages programmers

[19] See, for example, https://en.wikipedia.org/wiki/Computer_multitasking

to play a little fast and loose with the requirement that a task always remain responsive. ("I'll send off this message, I should get the reply almost immediately, it'll be OK to block briefly to read that reply". And then when the hardware hangs and the message doesn't come back at all, the whole task is dead to the rest of the world.)

The original DRAMA does provide a scheme for handling the problem of wanting to write sequential code in a sequential way[20], but this is an advanced topic, and not covered in this brief overview.

4.5 DRAMA messages

The fixed part of a DRAMA task is continually waiting for and reading DRAMA messages, usually from other tasks but occasionally, as in the case of messages generated by internal timers, from itself. Many of these messages are handled internally by the fixed part itself. The remainder are handled by those routines registered as the action handlers or the kick handlers for the various actions.

The action handler can be invoked for any of quite a large number of reasons, and not all will be listed here, given that this is supposed to provide just an overview of DRAMA. But the main reasons an action handler or kick handler can be invoked are:

Obey — The original entry to the handler, when the action has just been invoked.

Kick — The action has been kicked. An SDS structure, as supplied by the kicking task, provides details.

Reschedule — The action was waiting for a set time, and that time has now expired.

Signal — Code running in the current task has sent a signal to this action. This can have come from an interrupt handler, another action, or a separate thread of the task[21].

Complete — The action was waiting for a message, and this has now arrived.

---

[20] It involves the routine DitsActionWait().

[21] Even in the original DRAMA design, a task can make use of threads (usually using the POSIX thread library) independent of the main DRAMA thread, and these can send signals to actions run by the main thread.

4.6 Other DRAMA facilities

Amongst the things DRAMA allows an action to do are:

- Load another task, either on the local or on a remote machine.
- Establish a connection to another task, either local or remote.
- Invoke an action in another task. The progress of that action can be followed through trigger messages sent from that task, and a message sent on completion.)
- Kick an action it had previously invoked. (Which can include requesting that the action cancel itself.)
- Modify one of the current task parameters. (If any of these are being monitored by other tasks, these will be notified of the change.)
- Get the value of a specified parameter of another task.
- Set the value of a specified parameter of another task. (Not often used.)
- Output a message to the user. (The message is sent to the action that invoked the current action, and so on down the chain until it reaches a user interface of some sort, at which point it gets output. DRAMA is very careful that all messages are output in the context of an action, and are passed back to the ultimate invoker of that action.) A message can be flagged as an error message, in which case it can be displayed as such when output.
- Log a message to a log file kept by the task.
- Initiate a transfer of bulk data to another task, with a minimal copying of data. In order to make this as efficient as possible, the mechanism used for bulk data makes use of memory shared between processes and is different to that used for normal DRAMA messages.

4.7 Supporting Standards

DRAMA makes it easy to support standards across sets of tasks. Standard functionality can be inherited and the sub-classed as needed. For example, for AAT instrumentation, all tasks are required to provide a core set of Actions and Parameters. The default implementation is provided by package inherited by all tasks, but can be overridden as needed. This allows the higher-level control task to presume that all tasks implement this functionality, simplifying its design. In addition, any AAT instrumentation camera task obeys an additionally standard common to all camera tasks, with a default implementation provided. To implement a camera control task with allow required functionality, actions and parameters, requires implementing only

four methods (Initialise, Expose, Readout, Shutdown). This is all available in the C language interfaces.

5 DRAMA 2

There were two specific aims that drove the development of DRAMA2.

The original DRAMA design had evolved over time, and had resulted in a mixture of subroutine libraries, all designed to be called from C. Although these had a distinct hierarchy, it was not always clear which library provided which facility. A set of C++ classes had also evolved, which simplified writing DRAMA programs considerably, but the design of these reflected their evolutionary nature. It was felt that DRAMA needed a modern, coherent, C++ API. Moreover, the new C++11 and C++14 standards made it much easier to provide some of the facilities such a coherent API required.

Almost all systems on which DRAMA was now used supported multi-threading, and this was obviously the way to implement multiple concurrent actions within a task.

This means that both the code and structure of a program written for DRAMA2 look quite different to that of a program written for the original DRAMA design. Remarkably, however, the core of the DRAMA system needed only very minor changes to accommodate DRAMA2, and DRAMA2 really is just a new API for DRAMA rather than a new implementation of DRAMA.

Because this document is intended only as an introduction to DRAMA concepts, and as such as so far managed not to show any actual DRAMA code[22], the detailed changes to the API can be ignored here. The multi-threading aspects of the new system are important, and will be covered here, but the underlying concepts described so far are still important when it comes to understanding DRAMA2.

5.1 DRAMA2 actions

In DRAMA2, you can have actions that work just the same, staged, way as actions in the original DRAMA design. Such an action is embodied by an object

[22] Although there is no shortage of example code in the DRAMA documentation!

of a class that inherits from a base DRAMA 'message handler' class[23], and this merely provides a more modern interface to the old way of working.

Much more interestingly, you can also have actions that are handled by separate threads. Such actions are embodied by an object of a class that inherits from a base DRAMA 'threaded action', and provides a method that is called when the action is invoked[24]. See Figure 3

```
class Action1 : public thread::TAction{
public:
  Action1(std::weak_ptr<Task> theTask):
    TAction(theTask) {}
private:
  void ActionThread(const sds::Id &) override {
    // This is where the real work is done.
    MessageUser("Hello World - from a thread");
  }
};
```

Figure 3.DRAMA 2 Threaded action implementation

This method, since it runs in a separate thread, does not have to constantly return to the fixed part in order to keep the overall task responsive to incoming messages, and as a result it is free to do things that would have been seriously anti-social in the original DRAMA design, such as issuing blocking read calls, waiting for semaphores, even diving into long CPU-intensive calculations. It can do all of these without getting in the way of other actions, and the programmer can code a long sequential series of such operations just as a block of sequential code.

This is quite liberating!

There are also alternative approaches to implementing actions including simple functions and methods of various any desired class

However, there is always a 'however'; there is always at least a small catch. If such an action handler is blocking, waiting for some external event (which may never happen, of course), how do we interact with the action? How does a kick message get through to it? How do we cancel such an action?

---

[23] Since a DRAMA action handler is really just something that receives messages in the context of a specific action.

[24] The base class is drama::thread::TAction, and the method that handles invocation is called ActionThread().

(One might imagine cancelling an action by cancelling the thread, using something like pthread_cancel(), but this is a dangerous operation that often leaves resources in an uncertain state. Java deprecated its equivalent routine, and the C++ std::thread class does not provide such a routine directly[25]. Even more to the point, all you can do this way is cancel the thread; there's no way to implement more subtle interactions.)

So there is a price to pay for being allowed to block in this way in an action handler. It is that any such blocking operation must be done in such a way that it can be interrupted if necessary.

This is not a trivial thing to arrange, but DRAMA2 goes out of its way to make things easy.

The simplest case is where the action handler has done all it needs to for the moment and just wants to wait for a kick message to tell it what to do next. DRAMA2 provides a routine that simply waits for such a message, with an optional timeout.

More complex cases are where the action handler is doing something that is not so obviously something that DRAMA can be expected to deal with. For example, the action handler may simply make a read() call, which normally will not return until whatever it is trying to read becomes available.

What the action handler has to do is this. Just before it does something that will block, it creates a new object that inherits from a base 'kick notifier' class. This class starts a new background thread that can receive any kick messages. The kick notifier and the action handler code then have to work together to arrange to interrupt whatever blocking operation is taking place.

The simplest case would be a hard CPU loop which is able to make a test at regular points through the loop and bail out if necessary. The kick notifier provides an enquiry routine that can be used to ask if a kick has been received, and this could be used to abort such a CPU loop. See Figure 4. But that's the really easy case. A read() call is trickier.

The key is that there is a way to interrupt any given blocking, but different operations have to be interrupted in different ways. For example, a read() can be interrupted by sending a signal to the reading thread, or – depending on the driver – by closing the open file descriptor from another thread. The kick notifier has a 'kicked' method that gets invoked when a kick is received, and

---

[25] You may be able to access the underlying POSIX thread and use pthread_cancel() on it, but you'd be well advised to attempt nothing of the sort.

this can be overridden so that it interrupts the blocking operation in the action handler thread using such a technique. The read() will return with an error, and the action handler code can then check with the kick notifier to ask what happened.

```
void ActionThread(const drama::sds::Id &) override  {
    MessageUser("Action Starting");
    // Creates a another thread waiting for kicks.
    //   Destructor will clean it up.
    drama::thread::KickNotifier unblockObj(this);
    for (unsigned i = 0; i < 50 ; ++i)
    {
        for (unsigned j = 0; j < 100000000 ; ++j)
        {
        }
        MessageUser("Alive");
        if (unblockObj.WasKicked())
        {
            MessageUser("Action Was kicked.");
            return;
        }
    }
    MessageUser("Action complete");
}
```

Figure 4.DRAMA 2 Kickable hard loop

The point behind all this is that there is no single way that can interrupt all blocking operations cleanly, but DRAMA2 provides a way for the blocking thread to respond to a kick (in a sub-thread) and to interrupt its blocking action in the appropriate way.

Because threads are very lightweight items, and can be created very quickly as needed, they can be used very effectively in DRAMA2 programs to solve a variety of problems. A DRAMA2 action handler can split itself into a number of concurrent threads, and often does if it needs to communicate with a number of other tasks.

6 Summary

This introduction has only scratched the surface of what DRAMA provides, but it should have provided enough background to give an idea of what DRAMA can do.

For more information, there is a very detailed DRAMA2 manual available from AAO, and most of the documentation for the original DRAMA design is available on the web http://drama.aao.org.au/ or Tony Farrell (tony.farrell@mq.edu.au)

7 DRAMA Glossary

AAO	"Australian Astronomical Optics". Until 2010 the AAO was called the "Anglo-Australian Observatory", and then from 2010 to 2019, was "The Australian Astronomical Observatory".
AAO MQ	A department of Macquarie University which has taken over the instrumentation development roles of the old "Australian Astronomical Observatory" and is part of the "Australian Astronomical Optics" collaboration.
Action	A named operation that can be performed by a DRAMA task. It can have parameters associated with it through an SDS structure. It is invoked by sending an 'obey' message to the DRAMA task and is handled by a routine registered as the 'action handler' for that action. A DRAMA task can have a number of concurrent actions active.
Action handler	A user-written routine called first when a DRAMA action is invoked. In the original DRAMA design it is expected to complete almost immediately and return to the fixed part, which will call it again as required as the action progresses. Most user code in a DRAMA task takes place in action handlers. In DRAMA2 an action handler can run in a separate thread and does not need to keep returning to the fixed part.
ADAM	Astronomical Data Acquisition Monitor. A pre-cursor to DRAMA. Originally written at RGO for Perkin-Elmer minicomputers, and modified at ROE for VAX/VMS

	computer systems. Many of the concepts used in the original DRAMA design came from ADAM.
DRAMA	The AAO distributed data acquisition environment. DRAMA is not an acronym – it comes from "ADAM redone, about right", which only people who solve cryptic crosswords will follow, unfortunately.
DRAMA2	A modern API for DRAMA,making use of C++14 facilities, that supports multiple concurrent actions through multi-threading.
IMP	Inter-process Message Passing – the DRAMA sub-system used for inter-process communication.
Kick	A DRAMA message sent to an already running action, usually to influence the progress of that action. A common use of a kick is to cancel an action.
Kick handler	A user-provided routine called when a DRAMA action is kicked.
Monitor	A DRAMA task can request that it be notified whenever a specified parameter of a given task changes, in which case it is sent a copy of the parameter with its new value. This is commonly used by GUI tasks to display the state of the tasks for they provide an interface.
Obey	One of the types of message that can be sent to a DRAMA task. An obey message includes the name of an action to be invoked and an optional SDS structure containing named parameter values.
Parameter	Is used in two different ways in DRAMA. An action has named parameters passed in the SDS structure used to invoke it, which control the details of the action. A task has parameters, which are named SDS structures that are visible to other DRAMA tasks, and which can be monitored by such tasks.
SDS	Self-defining Data System. A subroutine library that allows arbitrarily complex structures containing named sub-structures and named primitive objects (integer or floating point values, either single values or arrays) to be created in memory and serialised either in memory or as

	disk files. It provides a full set of routines for searching and accessing such structures.
Task	Means what it means for most operating systems: a self-contained execution unit with its own address space, scheduled pre-emptively along with other tasks.
TCS	Telescope control system.
Trigger	A DRAMA message sent back from an executing action to the action that invoked it, usually as a progress report of some sort.
VxWorks	A commercial ‘hard’ real-time operating system, developed by Wind River.

Research Data and Software
Australian Astronomical Optics,
Macquarie University.
105 Delhi road, North Ryde, NSW 2113
T:+61 (02)9372 4800
http://drama.aao.org.au/
http://www.aao.gov.au
http://www.rds.org.au
tony.farrell@mq.edu.au