AUSTRALIAN ASTRONOMICAL OBSERVATORY
DRAMA Software Report 14
Version 1.24

Tony Farrell
1/09/2017

DRAMA 2

Contents

1 Introduction 7
1.1 DRAMA 2 – Hello World 7
1.1.1 Working through the example 8
1.1.1.1 Include file and DRAMA namespace 8
1.1.1.2 Action Definition 8
1.1.1.3 Task Definition 9
1.1.1.4 Main function 9
1.2 Example Code Location 9
1.3 Building Examples 9
1.4 Running the example 10
1.5 Documentation 10
1.6 Compilers and Operating Systems 10
2 Adding Actions and Parameters to a task. 11
2.1 A Simple EXIT action, functions for action implementations 12
2.2 Task Parameters 13
2.2.1 drama::Parameter 13
2.2.1.1 drama::Task::TaskPtr() 15
2.2.2 drama::ParSys 15
2.2.3 ditscmd and parameters. 17
3 DRAMA Status and C++ Exceptions 19
3.1 Catching and dealing with exceptions in main() 20
3.2 Exceptions in action handlers 22
3.3 More detail on drama::Exception 24
3.4 Exceptions within destructors 24
4 SDS and Command Arguments 25
4.1 SDS 25
4.1.1 Copying SDS Items 25
4.1.2 Constructing drama::sds::Id items 25
4.1.3 SDS – Data operations 27
4.1.3.1 Creating an SDS Structure 30
4.1.3.2 Inserting data into SDS. 30
4.1.3.3 Retrieving data from SDS 31
4.1.3.4 Other SDS Methods 31
4.1.3.4.1 Navigating Structures 31
4.1.3.4.2 Navigating structures using iterators 32
4.1.3.4.3 Viewing Structures. 32
4.1.3.4.4 Modifying Structures 32
4.1.3.4.5 Extra Data 33
4.1.3.4.6 Export/Import of structures 33
4.1.3.4.7 Export/Import of IDs 33
4.1.3.4.8 Copying 34
4.1.3.4.9 Direct access to the SDS Data. 34
4.1.3.4.10 SDS Compiler. 36
4.2 DRAMA Argument Structures 36
4.3 The `gitarg` namespace 37
4.4 Accessing action arguments 40
4.5 Returning SDS structures to the action's source talk 41
4.5.1 Trigger Messages 41
4.5.2 Action Completion Message Arguments 42

4.6 Complex parameters 42
4.6.1 Complex parameters with drama::Parameter 43
4.6.2 Complex parameters with drama::ParId 44
5 Action Rescheduling 45
5.1 Basic rescheduling. 45
5.2 Rescheduling after a delay. 46
5.3 Other reschedule reason codes. 47
5.4 Changing the Action Handler 47
5.5 Handling Kick Messages when Rescheduling. 48
5.6 Other ways of implementing handlers (e.g. as functions) 49
5.7 Spawnable Actions 51
6 Implementing Actions using Threads. 54
6.1 Major differences between pthreads and C++11 54
6.2 Implementing C++11 Threads in any Application 54
6.2.1 Futures and async. 55
6.2.2 Relationship to POSIX Threads. 55
6.3 Basic DRAMA2 Approach 55
6.4 Implementing DRAMA Actions using threads 56
6.5 Accessing Action Arguments 57
6.6 Kick Messages. 57
6.7 User initiated threads 58
6.7.1 POSIX Threads 60
6.8 Kicking threads that are blocked. 60
6.9 Trigger Messages, Output Arguments 61
6.10 Other ways of specifying handlers (e.g. as functions) 62
6.11 Locks - Working with older DRAMA interfaces or other shared data 63
6.12 Interactions with signals 63
6.13 Thread Programming Issues 64
6.13.1 Threaded Programs not exiting when expected. 64
6.13.2 Delayed Exception delivery 64
6.13.2.1 Delayed until an destructor is run 65
7 Sending Messages to tasks 66
7.1 DRAMA Task loading and Networking configuration 66
7.2 DRAMA 2 Message sending basics 66
7.3 Path constructor and related methods. 66
7.4 Loading tasks and Getting Paths 67
7.4.1 drama::Path methods that impact task loading. 69
7.4.2 drama::Path methods that impact Get Path operations. 69
7.5 Loading non-DRAMA programs. 70
7.6 Sending Obey Messages 70
7.6.1 Adding an argument to the Obey message 71
7.6.2 Recovering the completion message argument value from an Obey. 71
7.7 Sending Kick Messages 72
7.8 Changing how the methods respond to messages. 74
7.8.1 Responding to Trigger Messages 77
7.8.2 Responding to Kick Messages 78
7.8.3 Get Path messages 79
7.9 Message waits with timeouts 80
7.10 Orphaned Transactions 81
7.10.1 Default Behavior 81
7.10.2 Changing the default behavior 81
7.10.3 Actions taking over orphans 82
7.10.4 Creating an orphan on demand. 82
7.11 Parameter Set/Get Messages 82
7.12 Parameter Monitoring 83
7.12.1 Standard monitoring vs. forward monitoring 84

7.12.2 The Monitor Messages. ... *84*
7.12.3 Monitor to Parameters ... *85*
7.12.4 Monitor by Type ... *88*
7.12.5 Forward Monitors. ... *92*
7.12.6 Cancelling Monitors ... *92*
7.13 Control Messages 92
7.14 Multiple simultaneous messages from one action. 93
7.15 Sequencing issues. 98
8 GIT Task Implementation 100
8.1 Overriding GIT Action Implementations. 101
8.2 Accessing Simulation. 102
8.3 GIT POLL Action 102
8.4 GIT Path. 104
8.4.1 Initialise Method ... *104*
8.4.2 Exit Method ... *105*
8.4.3 Poll Method ... *105*
8.4.4 PollCancel Method ... *105*
8.4.5 Report Method ... *105*
8.4.6 Other Methods. ... *105*
8.4.7 Example Usage ... *106*
8.4.8 Sub classing git::Path ... *106*
9 Bulk Data 108
9.1 Sending Bulk Data 108
9.1.1 Creating a Bulk Data shared memory segment ... *108*
9.1.2 Sending a Bulk Data trigger message ... *109*
9.1.3 Sending a Bulk Data Obey/Kick message ... *113*
9.2 Receiving Bulk Data 114
9.2.1 Non-threaded actions ... *114*
9.2.2 Threaded Actions ... *116*
10 User Interfaces 117
11 Logging 121
11.1 Simplified Usage 121
11.2 Environment Variables 121
11.3 Log file locations/naming/day rollover. 122
11.3.1 Day Rollover ... *123*
11.4 Actions/Parameters 123
11.4.1 LOG_LEVEL Action ... *123*
11.4.2 Parameters ... *123*
11.4.3 Related control messages ... *123*
11.5 Logging levels 123
11.6 Opening the log file. 124
11.7 Logging Messages 125
11.7.1 SLog() 125
11.7.1.1 Limited formatting with SLog() via Manipulators ... 126
11.7.2 Log Stream Buffers ... *126*
11.8 Interaction with DITS debugging. 127
11.9 Log file content. 127
11.10 Type and thread safe printf style output to stdout/stderr. 128
11.10.1.1 Limited formatting control with SafePrintf() via Manipulators ... 129
12 Internals 131
12.1 Action Threads 131
12.1.1 ActionThreadComplete() ... *131*
12.1.2 ObeyReschedule() ... *131*
12.1.3 ProcessDrama2Signal() ... *131*
12.2 Sending Messages from threads 132
12.2.1 Type message – Obey() ... *132*
12.2.1.1 Send() 132

12.2.1.1.1 SetupWaitEvent() *132*
12.2.1.2 WaitForTransaction() 132
12.2.2 ProcessSubsidiaryMessage *133*
12.3 Dealing with shutdown. 133
12.3.1 Normal Action Thread Shutdown *133*
12.3.2 Normal UFACE Thread Shutdown *134*
12.3.3 Possible Flaws *134*
12.3.3.1 UFACE Case. 134
12.3.3.2 Action Case 134

Table of Examples

Example 1–1. Hello World in DRAMA 2 7
Example 2–1. Different ways of adding actions 11
Example 2–2. Example of the use of drama::Parameter 13
Example 2–3. Use of drama::ParSys 15
Example 2–4. Use of ditscmd with parameters 17
Example 3–1. Catching drama:Exception in main() 20
Example 3–2. Catch all exceptions. 22
Example 4–1. SDS Usage example 27
Example 4–2. Accessing SDS data via pointers 35
Example 4–3. Constructing Arg style SDS structures 36
Example 4–4. Reading a structure with gitarg (for source - see example 4.3 source) 38
Example 4–5. Accessing message arguments 41
Example 4–6. Sending Trigger messages 41
Example 4–7. Setting a completion message argument 42
Example 4–8. Complex parameters with the drama::Parameter class 43
Example 4–9. Complex parameters with drama::ParId. 44
Example 5–1. Basic Rescheduling 45
Example 5–2. Rescheduling after a delay 46
Example 5–3. Changing Obey handlers 47
Example 5–4. Kick Handlers 48
Example 5–5. Specifying functions/methods as handlers 50
Example 5–6 Spawnable Action Example 52
Example 6–1. Implementing an action with a thread 56
Example 6–2. Threaded action argument 57
Example 6–3. Kicking a threaded action 58
Example 6–4. Theaded action with child thread 58
Example 6–5. Kicking a blocked thread. 61
Example 6–6. Implementing a thread action with a function. 62
Example 7–1. Loading tasks and getting the path 68
Example 7–2. Sending an Obey message 70
Example 7–3. Sending an Obey with an argument. 71
Example 7–4. Accessing Obey message completion argument 71
Example 7–5. Sending kick messages 72
Example 7–6. Handling trigger messages 77
Example 7–7. Handling kick messages whilst waiting for subsidiary message 79
Example 7–8. Obey with timeout. 80
Example 7–9. Getting and setting parameters 82
Example 7–10. Monitoring to parameters 86
Example 7–11. Monitoring by type 88
Example 7–12. Sending multiple messages from one action thread 94
Example 7–13. Monitor Kick Handler - MonitorMessageHandler 95
Example 7–14 Monitor Kick Handler – Action Code. 96
Example 7–15. Monitor Kick Handler – processing the kick. 97
Example 7–16 Monitor Kick Handler –sending the Monitor Kick. 97
Example 8–1. A Basic GIT Task. 100
Example 8–2. Overriding GIT Action Implementations 101
Example 9–1. Sending a bulk data trigger message 110
Example 9–2. Obey message with bulk data. 113
Example 9–3 Receiving Bulk Data 115
Example 10–1. Basic User Interface Example 118

1 Introduction

The AAO DRAMA Software is a set of API's and tools used to develop distributed software capable of soft real-time performance. DRAMA has been around for quite some time – having originally been developed under the C language, prior to ANSI C's certification and a long way before the standardization of C++. The C++ interfaces (SDS Library, ARG Library, Dcpp and various GIT interfaces) have been added in a haphazard fashion over the years and Dcpp in particular is very poorly designed.

Additionally – the POSIX Thread library is now common to architectures of interest and it is attractive to use threads to implement actions.

The 2011 version of the C++ Standard (C++11) has now been released and is well supported by compilers of interest. It provides many useful new features and standard support for threads and other modern features.

This document introduces DRAMA 2, a modern interface to DRAMA using much of the power of C++11. It allows DRAMA actions to be implemented in threads, reducing the complexity of many tasks.

1.1 DRAMA 2 – Hello World

So what does a DRAMA 2 program (known as a "task") look like? In the style of the traditional "Hello World" program, the simplest DRAMA program will accept just one DRAMA Action (Action = Command), output a message and cause the program to exit. Example 1–1 gives such an example in DRAMA2.

Example 1–1. Hello World in DRAMA 2

```
1.  #include "drama.hh"
2.
3.  // Action definition.
4.  class HelloAction : public drama::MessageHandler {
5.  public:
6.      HelloAction() {}
7.      ~HelloAction() {}
8.  private:
9.      drama::Request MessageReceived() override {
10.         MessageUser("Hello World - from DRAMA 2");
11.         return  drama::RequestCode::Exit;
12.     }
13. };
14.
15. // Task Definition
16. class DramaExampleTask : public drama::Task {
17. private:
18.     HelloAction HelloActionObj;
19. public:
20.     DramaExampleTask(const std::string &taskName) :
21.         drama::Task(taskName) {
22.         Add("HELLO",
23.             drama::MessageHandlerPtr(&HelloActionObj,
24.                                      drama::nodel()));
25.     }
26. };
27. // Main program.
28. int main()
29. {
```

```
30.      DramaExampleTask task("EXAMPLE1_1");
31.      task.RunDrama();
32.      return 0;
33.  }
```

As per most example code in this document, the example has line numbers prefixed to each line. Additionally, the task name indicates the example within the section, so EXAMPLE1_1 is section 1 (this section), example 1. EXAMPLE1_2 would be the second example in this section (if there was one).

1.1.1 Working through the example

The example program creates a DRAMA task named "`EXAMPLE1_1`", which has one and one only action – "`HELLO`". The "`HELLO`" action will output a message to the user and then cause the task (program) to exit.

1.1.1.1 Include file and DRAMA namespace.

At line #1, "drama.hh" is included. This include file will pick up all of the DRAMA facilities needed in a basic program.

All the DRAMA2 facilities are found within the "**drama**" namespace. As a matter of policy, the author does not use "`using namespace…`" declarations in source files are not part of implementing the namespace in question, so in these examples, each name from the "drama" namespace will always be prefixed by "`drama::`".

1.1.1.2 Action Definition

Line #4 is the definition of the class "`HelloAction`", which will be used to implement the "`HELLO`" action (command). This class implements the "`drama::MessageHandler`" interface. Action implementations and DRAMA reschedule event handlers must implement this interface and implement the "`MessageReceived`" method. That method will be invoked when the Action is "Obeyed" (That is, when a message of type "Obey", specifying the "action" name is sent to the task.).

Line #9 is the beginning of the implementation of "`drama::MessageHandler`". Note the use of the "`override`" specification. This C++11 keyword indicates we are intending to override a method in the sub-class and the compiler should complain if we are not. Please use this keyword whenever overriding a method, as it ensures simple mistakes like errors in the method name spelling or the argument list are picked up.

The "`drama::MessageHandler`" class provides "`MessageUser()`", a method for sending messages to the user interface, equivalent to DRAMA's traditional `MsgOut()` call. At line #10, this is used to send our "Hello World" message. As an alterative, there is also the `drama::MessageUserStreamBuf` class, which can be used to contruct a `std::ostream` sub-class that can be used to output such messages. This is how this might be used:

```
drama::MessageUserStreamBuf<decltype(*this)> messStreamBuf(*this);
std::ostream messStream(&messStreamBuf);
messStream << "First line for message stream" << std::endl;
```

The method return at line #11 is used to specify some result of the action. In this case, we are requesting that the task Exit when the method completes, by returning "`drama::RequestCode::Exit`". Various other values are possible, and will be described later in the document.

1.1.1.3 Task Definition

In DRAMA 2, the "`drama::Task`" class is used to implement a DRAMA task. The implementer must create one of these objects, add DRAMA Actions and Parameters to the task and then invoke the `RunDrama`() method to process DRAMA messages.

The normal approach is to sub-class "drama::Task", as in done at line #16 of Example 1–1, with the class named `DramaExampleTask`. At line #18, the object "`HelloActionObj`", implementing the HELLO action, is defined.

At line #20, we see the task constructor. It passes the user specified DRAMA task name to the "`drama::Task`" constructor.

In the body of the constructor, at line #22, the `drama::Task Add()` method is invoked to create the "HELLO" action. The first argument is the action name; the second is a shared pointer to the object that will implement the action – the `HelloActionObj` object. The shared pointer is created using the `drama::MessageHandlerPtr()` constructor, specifying the address of the object and `drama::nodel()` as the constructors second argument, indicating the object should not be deleted when the shared pointer is destroyed. We will examine the `Add()` method in more detail later.

1.1.1.4 Main function

In the main function, from line #27, we create the task and then run it. The `drama::Task RunDrama()` method will return when an action indicates the task should exit.

It should be noted that the DRAMA Task is registered with DRAMA when `drama::Task` constructor is invoked. The `RunDrama()` method is the message processing loop – you invoke that method to actually start the task processing DRAMA messages..

1.2 Example Code Location

All example code from this document is found in the "Drama2Examples" Sub-system (ACMM Module Drama2Examples). There is a sub-directory for each section of this document and files are named after the example number. So Drama2Examples/sec1/exam1_1.cpp contains the code from Example 1–1.

1.3 Building Examples

To build these examples, you need a DRAMA installation, with the new Drama2 sub-system installed (ACMM Module Drama2). The "Drama2Examples" sub-system contains a DRAMA "dmakefile" that can build all the examples using:

```
cd Drama2Examples
dmkmf -g
make
```

You can examine the dmakefile to see how this is done. The one thing unusual about this dmakefile is that the example source files are in sub-directories of Drama2Examples, which requires extra lines in the dmakefile to build the object files.

1.4 Running the example

You can send the "HELLO" action (command) to this task using any general DRAMA user interface, of which there are a few. "`ditscmd`" is the simplest way

```
>> ./exam1_1 &
>> ditscmd EXAMPLE1_1 HELLO
DITSCMD_10ff:EXAMPLE1_1:Hello World - from DRAMA 2
[3]  + done       ./exam1_1
```

1.5 Documentation

The detailed documentation for the classes, types and functions of DRAMA2 are generated from the source code using the "DOXYGEN" tool. Please see <<web site>>. For DRAMA itself, please see <<drama-web-site>>

1.6 Compilers and Operating Systems

Due to it requirement for C++11 features, DRAMA 2 requires a modern compiler. GCC 4.8.2 or later is needed (beware of a serious bug in GCC 4.9.0), and GCC is the standard complier for Linux.

For Mac OS X – GCC 4.8.2 should work (TBC), but Apple X-Code now provides LCC rather then GCC.

OS	Version	Compiler	Version
Linux		GCC	4.8.2 +
Mac OS X		GCC	TBC
Mac OS X	Lion		TBC
Mac OS X	Mt. Lion	XCode-LCC	TBC
Mac OS X	Mavericks	XCode-LCC	TBC

2 Adding Actions and Parameters to a task.

`The drama::Task::Add()` method is used to add actions to a task. The Action implementation object is a sub-class of `drama::MessageHandler`, and must be specified as a shared pointer (`drama::MessageHandlerPtr()` to the object in question, to ensure it is not deleted until DRAMA is finished with it.
`drama::MessageHandlerPtr()` is declared as "`std::shared_ptr<MessageHandler>`".

Example 2–1 is a rework of the "Hello World" program from Example 1–1, showing the different ways an action can be added. The original code is at line #23, where we are specifying the address of a of `drama::MessageHandler` declared as part of the task. Here we must pass this address via a `drama::MessageHandlerPtr()` shared pointer, but need make sure the object is not deleted when the shared pointer releases it, since it is part of the task object. That is done by adding the optional "deleter" argument "`drama::nodel()`" to the `drama::MessageHandlerPtr()` constructor. See documentation for `std::shared_ptr` for more details on this "delete" argument.

Example 2–1. Different ways of adding actions

```
1.  #include "drama.hh"
2.
3.  // Action definition.
4.  class HelloAction : public drama::MessageHandler {
5.  public:
6.      HelloAction() {}
7.      ~HelloAction() {}
8.  private:
9.      drama::Request MessageReceived() override {
10.         MessageUser("Hello World - from DRAMA 2");
11.         return  drama::RequestCode::End;
12.     }
13. };
14.
15. // Task Definition
16. class DramaExampleTask : public drama::Task {
17. private:
18.     HelloAction HelloActionObj;
19. public:
20.     DramaExampleTask(const std::string &taskName) :
21.         drama::Task(taskName) {
22.
23.         Add("HELLO",
24.             drama::MessageHandlerPtr(
25.                 &HelloActionObj,
26.                 drama::nodel()));
27.         // Standard simple EXIT action.
28.         Add("EXIT", &drama::SimpleExitAction);
29. );
30.     }
31. };
32. // Main program.
33. int main()
34. {
35.     DramaExampleTask task("EXAMPLE2_1");
36.
37.     // Try adding actions from here, using a dynamically
```

```
38.      //   allocated Obey message handlers.  In all cases,
39.      //   the handler object will be deleted by DRAMA
40.      //   when the task shuts down.
41.
42.      // Approach using the new operator.
43.      task.Add("HELLO2",
44.               new HelloAction());
45.      // Approach using std::make_shared, may be more efficient.
46.      task.Add("HELLO3",
47.               std::make_shared<HelloAction>());
48.      // Can also be done this way.
49.      task.Add("HELLO4",
50.               drama::MessageHandlerPtr(new HelloAction) );
51.
52.      task.RunDrama();
53.      return 0;
54.  }
```

From line #43, we see various examples of adding actions using dynamically allocated objects, actions HELLO2, HELLO3 and HELLO4. Here we are using objects of the same "`HelloAction()`" class, but you could of course use any sub-class of `drama::MessageHandler`. Line #43 shows the most naïve example, specifying "`new HelloAction()`".to create the object. This does work, there is an overload of `Add()` which accepts this and created a shared pointer to the object. But line #46 is the preferred approach, as there is some scope for this to be implemented more efficiently by the compiler. Line #49 is yet another version.

2.1 A Simple EXIT action, functions for action implementations

One additional change in Example 2–1 compared to Example 1–1 is the return value of `HelloAction::MessageReceived()`. Rather then "`drama::RequestCode::Exit`", it now returns "`drama::RequestCode::End`" (line #11). This request says that the action has completed, but that the task should continue to run (accepting more messages). This was done to allow testing of the various different forms of the HELLO action. But now we need a separate action to allow the task to shutdown correctly. Traditionally, this is done with an action named EXIT.

Whilst a task may want to do various things when it receives an EXIT action, and can sub-class `drama::MessageHandler` to implement what it requires, many simple tasks, such as our examples, just need the task to shutdown immediately. As a result, a simple implementation of an EXIT action class has been provided. This is the "`drama::SimpleExitAction`" function, which is specified as the EXIT action's implementation at line #28.

This demostates it is also possible to specify a fuction to handle your action, rather then an object. The argument to Add() in this case is of the type `drama::MessageReceiveFunction`. This is declared as

```
std::function<Request (MessageHandler *)>
```

By use of `std::function<>` and/or `std::bind()`, there are many ways to generate one of these. An object which is a sub-class of drama::MessageHandler is created, and invokes this function to handle messages, and the address of the object is passed as the first argument. This will be needed for any case where would might other call a method of the drama::MessageHandler class.

2.2 Task Parameters

DRAMA Tasks can have a "Parameter System" and most significant tasks will. Parameters are externally accessible named items that provide access to the task's state and can be used to set its configuration. Parameters have names and can have scalar values or complex values (structures, arrays etc).

DRAMA Provides messages to Set and Get parameter values in other tasks, and to monitor changes in their values. For example, a telescope control task will typically have parameters for the telescope position, current time, current coordinate system etc. A user interface for the telescope control task will monitor these parameters, which ensures it is notified when the values change. It will then update displayed values. This monitoring of parameters approach ensures efficient user interface implementation, and allows multiple user interfaces to the one task to run at the same time.

The task implementing the parameters has the flexibility to implement parameters as they prefer, but all known DRAMA tasks use the "Simple DITS Parameter System" – SDP, to implement parameters. DRAMA2 presumes the use of SDP and provides a high level interface to it.

It should be noted that SDP Parameters are effectively a global variable of the task. So in cases where their value impacts the running of a task (a configuration parameter), they should be used with care and these cases should be avoided. The preferred use of parameters is as output only items (e.g. items intended for display in user interfaces).

DRAMA 2 implements two interfaces to the task's own parameter system. The "`drama::Parameter`" template class provides a simple interface which creates parameters and objects that transparently read/write parameter values. Alternatively, objects of the "`drama::ParSys`" class can be created and used to access parameters. This later class is independent of how the parameter was created and hence can be used in cases that "`drama::Parameter`" cannot.

2.2.1 drama::Parameter

Example 2–2 demonstrates the use of `drama::Parameter`. This task creates four scalar parameters, PARAM1 to PARAM4, each of a different type. Note that we generally consider a string parameter to be scalar, through its implementation is that of an array of char.

Example 2–2. Example of the use of drama::Parameter

```
1.  #include "drama.hh"
2.
3.  // Action definition.
4.  class HelloAction : public drama::MessageHandler {
5.  public:
6.      HelloAction() {}
7.      ~HelloAction() {}
8.  private:
9.      drama::Request MessageReceived() override {
10.         MessageUser("Hello World - from DRAMA 2");
11.         return  drama::RequestCode::End;
12.     }
13. };
14.
15. // Task Definition
16. class DramaExampleTask : public drama::Task {
17. private:
```

```
18.      HelloAction HelloActionObj;
19.  public:
20.      /*
21.       * Create task parameters
22.       */
23.      drama::Parameter<INT32> param1;
24.      drama::Parameter<std::string> param2;
25.      drama::Parameter<float> param3;
26.      drama::Parameter<UINT32> param4;
27.      /*
28.       * Constructor.
29.       */
30.      DramaExampleTask(const std::string &taskName) :
31.          drama::Task(taskName),
32.          param1(TaskPtr(), "PARAM1", 2),
33.          param2(TaskPtr(), "PARAM2", "hi there c++"),
34.          param3(TaskPtr(), "PARAM3", 33.3),
35.          param4(TaskPtr(), "PARAM4", 4) {
36.
37.          Add("HELLO",
38.              drama::MessageHandlerPtr(
39.                  &HelloActionObj,
40.                  drama::nodel()));
41.          // Standard simple EXIT action.
42.         Add("EXIT", &drama::SimpleExitAction);
43.
44.          double val = param3;
45.          std::cout << "On startup, parameter 3 has value "
46.                    << val << std::endl;
47.          param3 = 4.5;
48.          std::cout << "Parameter 3 now has value "
49.                    << param3 << std::endl;
50.      }
51.  };
52.  // Main program.
53.  int main()
54.  {
55.      DramaExampleTask task("EXAMPLE2_2");
56.      task.RunDrama();
57.      return 0;
58.  }
```

From line #23 to line #26, variables for each parameter are declared. The template argument must be an SDS compatible scalar type[1] or std::string. Note that use of INT32 for a 32 bit integer, rather then say “int” or “long”, since the length of those types varies. If you want a 64 bit type, use INT64.

The constructors for these variables are invoked at lines 32 to 35. A pointer to the task is requires, the name of the parameter and its initial value. Once the items are constructed, the parameters are ready for use.

Line #44 shows how you can fetch the value of the parameter and line #47 shows how to change it. That is, they can be used as if they are items of the scalar type they are representing.

[1] SDS Scalar types -> `signed char, unsigned char, unsigned short, short, INT32, UINT32, INT64, UINT64, float, double`.

2.2.1.1 drama::Task::TaskPtr()

A slight division here. In Example 2–2, we see at lines 31 to 35 the use of the `drama::Task::TaskPtr()` method. This method is used to retrieve a `std::weak_ptr` to the Task object. Any DRAMA2 method which might store away details on the drama::Task object will want a `std::weak_ptr<drama::Task>` object, to ensure pointer use safety. Any attempt by these methods to run after the task has been destroyed will result in an exception (rather then a segmentation volation core dump). It is recommend that you pass pointers to task objects, when required, using `std::weak_ptr<drama::Task>`.

Before using them, you must convert to a `std::shared_ptr<drama::Task>`. The constructor will throw if the weak pointer is invalid. Many of DRAMA2 classes have a `GetTask()` method which returns a `std::shared_ptr<drama::Task>` for the task that were created with.

You can then use `drama::TaskPtrAs()` to downcast this to your own sub-class of `drama::Task`. E.g.

```
class MyTaskClass :: public drama::task {
…
void MethodOfSomeClassWithGetTask
{
    auto myTask(GetTask()->TaskPtrAs<MyTaskClass>());
    myTask->MyMethod();'
}
```

2.2.2 drama::ParSys

There is an issue in Example 2–2. In order for say the "`MessageReceived`" method to access the parameter value, the object must be available to it. There are various ways to achieve this, particularly in this simple case. But for complex cases where the method accessing the parameter is deep compared to the task, this could be a problem. `drama::ParSys` provides an alterative. It makes the entire parameter system of a task easily available. Example 2–3 shows a reimplementation of the `MessageReceived()` method of Example 2–2 demonstrating the use of `drama::ParSys`.

Example 2–3. Use of drama::ParSys

```
1.   drama::Request MessageReceived() override {
2.           drama::ParSys parSys(GetTask());
3.
4.           bool bVal=false;
5.           char   cVal=0;
6.           short sVal=0;
7.           unsigned short usVal=0;
8.           INT32  iVal=0;
9.           UINT32 uiVal=0;
10.          INT64 i64Val=0;
11.          UINT64 ui64Val=0;
12.          double dVal=0;
13.          float  fVal=0;
14.          std::string strVal;
15.
16.          parSys.Get("PARAM1", &bVal);
17.          parSys.Get("PARAM1", &cVal);
18.          parSys.Get("PARAM1", &sVal);
```

```
19.          parSys.Get("PARAM1", &usVal);
20.          parSys.Get("PARAM1", &iVal);
21.          parSys.Get("PARAM1", &uiVal);
22.          parSys.Get("PARAM1", &i64Val);
23.          parSys.Get("PARAM1", &ui64Val);
24.          parSys.Get("PARAM1", &fVal);
25.          parSys.Get("PARAM1", &dVal);
26.          parSys.Get("PARAM1", &strVal);
27.
28.          std::cout << "::Get()s of PARAM1"
29.                    << ": Bool = "    << bVal
30.                    << ", Char = "    << (int)(cVal)
31.                    << ", Short = "   << sVal
32.                    << ", U Short = " << usVal
33.                    << ", INT32 = "   << iVal
34.                    << ", UINT32 = "  << uiVal
35.                    << ", INT64 = "   << i64Val
36.                    << ", UINT64 = "  << ui64Val
37.                    << std::endl
38.                    << "  Float = "   << fVal
39.                    << ", Double = "  << dVal
40.                    << ", String = "  << strVal
41.                    << std::endl;
42.
43.          std::cout << "Simple gets - PARAM3 (except string)"
44.                    << ": INT64 = "
45.                    << parSys.GetLong("PARAM3")
46.                    << ", UINT64 = "
47.                    << parSys.GetULong("PARAM3")
48.                    << ", Double = "
49.                    << parSys.GetDouble("PARAM3")
50.                    << ", String = \""
51.                    << parSys.GetString("PARAM2")
52.                    << "\"" << std::endl;
53.
54.          parSys.Put("PARAM1", (bool)true);
55.          parSys.Put("PARAM1", (char)2);
56.          parSys.Put("PARAM1", (short)3);
57.          parSys.Put("PARAM1", (unsigned short)4);
58.          parSys.Put("PARAM1", (INT32)(-5));
59.          parSys.Put("PARAM1", (UINT32)6);
60.          parSys.Put("PARAM1", (INT64)7);
61.          parSys.Put("PARAM1", (UINT64)(8));
62.          parSys.Put("PARAM3", (float)(9.1));
63.          parSys.Put("PARAM3", (double)(10.2));
64.          parSys.Put("PARAM3", 11.2);
65.          parSys.Put("PARAM2", "Result of char * put");
66.          parSys.Put("PARAM2",
67.                     std::string("Result of std::string put"));
68.
69.          std::cout << "Have completed all basic put operations"
70.                    << "  parameters look like:"
71.                    << std::endl;
72.
73.          MessageUser("All my parameter work is done");
74.          return  drama::RequestCode::End;
75.       }
```

A `drama::ParSys` item is constructed at line #2 and is then used to fetch and set the value of various parameters. The `drama::ParSys::Get()` method allows a parameter of a given name to be fetched as a specified type, with conversions done as required (so you fetch in the type you want, not needing to know the parameter's actual type – as long as the conversion is sensible). Similarly, the `drama::ParSys::Put()` method can be used to set the value of a parameter from any source type which is compatible. There are also methods like `drama::ParSys::GetLong()` which return a parameter's value directly, but only a few standard return types are supported (long, unsigned-long, double, string).

Hou can also use `drama::ParSys` to create parameters.
`drama::ParSys::Create()` allows the creation of a primitive parameter.
`drama::ParSys::CreateItem()` allows an SDS item to be inserted into the parameter system whilst `drama::ParSys::CreateSds()` creates an item of a given SDS type (typically a structure) and returns an drama::sds::Id item to allow you build a complex parameter and access it efficiently.

2.2.3 ditscmd and parameters.

The "`ditscmd`" program can set and get parameter values. If the option "`-g`" is supplied, then instead of specifying the action name as the second argument, you specify the name of a parameter to fetch. Subsequent arguments are the names of other parameters to fetch.

If you specify the "`-s`" option to "`ditscmd`", then you are setting the value of a single parameter. The second argument is the parameter name and the third is the new value for that parameter.

Finally, there are two special parameter names, "`_NAMES_`" and "`_ALL_`". When you specify "`_NAMES_`" as the parameter name in a "get" message, the return value provides the names of all parameters in the target task. If you specify the name "`_ALL_`" then the entire parameter system is returned in one message.

Example 2–4 shows the use of `ditscmd` with parameters.

Example 2–4. Use of ditscmd with parameters

```
>> ./exam2_3 &
>> ditscmd -g EXAMPLE2_3 PARAM1
DITSCMD_4b52:2
>> ditscmd -g EXAMPLE2_3 PARAM1 PARAM2
DITSCMD_4b53:2 hi there c++
>> ditscmd -g EXAMPLE2_3 PARAM1 PARAM2 PARAM3
DITSCMD_4b6e:2 hi there c++ 33.3
>> ditscmd -s EXAMPLE2_3  PARAM1 23
>> ditscmd -g EXAMPLE2_3 PARAM1
DITSCMD_4b72:23
>> ditscmd -g EXAMPLE2_3  _NAMES_
DITSCMD_4b78:Parameter names in task EXAMPLE2_3 (6 parameters):
DITSCMD_4b78:   LOG_LEVEL
DITSCMD_4b78:   GITLOG_FILENAME
DITSCMD_4b78:   PARAM1
DITSCMD_4b78:   PARAM2
DITSCMD_4b78:   PARAM3
DITSCMD_4b78:   PARAM4
>> ditscmd -g EXAMPLE2_3  _ALL_
DITSCMD_4b79:SdpStructure         Struct
DITSCMD_4b79:  LOG_LEVEL            Char   [5] "NONE"
DITSCMD_4b79:  GITLOG_FILENAME      Char   [1] ""
```

```
DITSCMD_4b79:  PARAM1              Int     23
DITSCMD_4b79:  PARAM2              Char   [13] "hi there c++"
DITSCMD_4b79:  PARAM3              Float   33.3
DITSCMD_4b79:  PARAM4              UInt    4
```

3 DRAMA Status and C++ Exceptions

DRAMA has traditionally implemented an "inherited status" approach to management of error conditions. Every routine had, normally as its last argument, a variable named "`status`" of type "`StatusType *`". On entry, the routine must check the value pointed to and if it is non-zero (`!= STATUS__OK`), it should return immediately. Any routine encountering an error should set status to a bad value and return. This leads to code like this:

```
void Routine(StatusType *status)
{
    if (*status != STATUS__OK) return;
    routine1(status);
    routine2(status)
    routine3(status);
    ….
}
```

This avoided typical C code that was a bunch of nested checks on possible returns from functions. Such an approach can work well in C and Fortran code. `StatusType` is a 32 bit integer and bad status values are defined using the DRAMA "`messgen`" tool such that each value can be unique, `#define` macros for the status code values are available as string translations of each status value.

The status values can be passed between DRAMA tasks, so a user interface can determine for example, why an action in a subsidiary task failed and output an appropriate textual error message.

DRAMA allows additional textual context to be added when errors occur using the "ERS" library. This allows a set of textual information to be put together into an error report that can be sent to the user interface. There is always a DRAMA Status associated with an Ers report, but there is not always an Ers report associated with a bad DRAMA Status value. Textual reports are important to allow things such as file names to be returned to the user .e.g "Failed to open file /dev/device".[2]

The DRAMA Status approach can go wrong if the programmer does not correctly check for bad status on entry to a routine, or does not correctly translate a failure in a routine without DRAMA Status (e.g. C RTL) to a DRAMA Status code. But other then these issues, it been proven to work well.

In C++ it is more natural use exceptions to handle errors. Exceptions can implement most of what is provided by the DRAMA Status and the ERS Library. Older DRAMA C++ interfaces did not attempt to use exceptions as they were poorly supported by compilers at that point in time, but DRAMA 2 can presume they are available and working well.

But C++ exceptions cannot be transmitted easily between tasks via DRAMA Messages. If an action throws an exception that is not caught in the action method, DRAMA 2 needs to convert it to a DRAMA Message that can be sent to the user interface or parent task for processing or display.

The following approach is used.

[2] The DRAMA Status codes are based on Digital VAX/VMS Status codes, with the inherited status convention itself inherited from the Starlink ADAM environment. The ERS library is based on the similar ERR library in the Starlink ADAM environment.

1. A `drama::Exception` class has been created. All error conditions generated in DRAMA code (DRAMA2 itself or user DRAMA code) should throw an object of this class or a sub-class of `drama::Exception`.
2. Each time DRAMA 2 calls an older style method, with a DRAMA Status variable, it checks the result after the call. If Status has gone bad, a "`drama::Exception`" object is created and thrown.
3. At any point where DRAMA style code with an inherited status argument calls C++ DRAMA 2 code, it must catch any `drama::Exception` which is thrown and convert it to a combination of a bad status value and ERS reports.

Most of the time, the implementer of a DRAMA2 task does not need to worry about the above, but they do need to consider when to catch and handle exceptions.

3.1 Catching and dealing with exceptions in main()

In the various examples looked at so far, no attempt is made to handle any exceptions in the `main()` routine. Exceptions could occur in the creation of the DRAMA task and the running of the task. It is easy to demonstrate exceptions of each of these types. For example, if you start one of the tasks and then use the DRAMA "taskclose" command to rudely shut it down (taskclose causes the message loop to exit without any user code being invoked), `task.RunDrama()` will throw an exception:

```
>> ./exam1_1 &
>> taskclose EXAMPLE1_1
Closedown request sent to task 'EXAMPLE1_1'.
terminate called after throwing an instance of 'drama::Exception'
  what():  DRAMA Task "EXAMPLE1_1" main loop exited with error
EXAMPLE1_1:exit status:%DITS-F-SIGABRT, DITS Exited via exit handler with
signal SIGABRT
 [3]  + abort (core dumped)  ./exam1_1
```

Alternatively, if you start a second copy of a one of these programs, then an exception is thrown when the task is created, due to a task of the same name already being registered:

```
>> ./exam1_1&
[3] 25749
>> ./exam1_1&
[4] 25750
>> terminate called after throwing an instance of 'drama::Exception'
  what():  task.cpp:68:DRAMA Task Initialisation failure
##EXAMPLE1_1: Cannot register task 'EXAMPLE1_1'. Task name already in use.
# EXAMPLE1_1:ImpRegister failed
EXAMPLE1_1:exit status:%DITS-F-SIGABRT, DITS Exited via exit handler with
signal SIGABRT

[4]  + abort (core dumped)  ./exam1_1
```

In both cases, the result is an abort and core dump, as with any uncaught exception.

Example 3–1 shows catching "drama::Exception" and reporting details from it.

Example 3–1. Catching drama:Exception in main()

```
1.   int main()
2.   {
3.       try
4.       {
5.           DramaExampleTask task("EXAMPLE3_1");
```

```
6.          task.RunDrama();
7.      }
8.      catch (drama::Exception &e)
9.      {
10.         // Note:need to have included <iomanip> for std::hex
11.         std::cerr << "drama::Exception caught by main()"
12.                   << std::endl
13.                   << e.toString()
14.                   << std::endl
15.                   << "DRAMA Status = 0x"
16.                   << std::hex
17.                   << e.dramaStatus()
18.                   << ", "
19.                   << e.dramaStatusStr()
20.                   << std::endl;
21.
22.         exit (e.statusAsSysExitCode());
23.     }
24.     return 0;
25. }
```

Line #11 to line #19 take advantage of features of `drama::Exception` to provide higher quality output then the standard exception. Note that at line #17 we are accessing the DRAMA status code associated with cause of the exception, which we can convert to a string using `drama::Exception::dramaStatusStr()`.

Line #22 converts the DRAMA Status code to a value appropriate for use with `exit()`.

Given the above code, the two examples now return a more informative error and does not core dump for the above two errors

```
>> ./exam3_1&
[3] 31216
>> taskclose EXAMPLE3_1
Closedown request sent to task 'EXAMPLE3_1'.
>> drama::Exception thrown and caught by main()
task.cpp:175:DRAMA Task "EXAMPLE3_1" RunDrama() exited due to error
DRAMA Status = 0xfd08114, %DITS-F-IMPSHUTDOWN, An Imp shutdown message was
received
[3]  + exit 4     ./exam3_1


>> ./exam3_1&
[3] 31244
>> ./exam3_1
drama::Exception thrown and caught by main()
task.cpp:68:DRAMA Task Initialisation failure
DRAMA Status = 0xf35802a, %IMP-E-DUP_TASK_NAME, Task name already in use
```

`drama::Exception` is a sub-class of `std::exception`, which does allow many standard try/catch blocks to catch it. But often you would want to implement a more complex catch block set in `main()` to be sure you know what is going wrong. See Example 3–2 for how this might be done. This example also shows how you can use a `drama::Exception` object as a stream output source directly (line 12), this line outputs the same information as lines 13 to 19 above.

Example 3–2. Catch all exceptions.

```
1.   int main()
2.   {
3.       try
4.       {
5.           DramaExampleTask task("EXAMPLE3_2");
6.           task.RunDrama();
7.       }
8.       catch (drama::Exception &e)
9.       {
10.          std::cerr << "drama::Exception caught by main()"
11.                    << std::endl
12.                    << e   // Outputs all the exception details.
13.                    << std::endl;
14.
15.          exit (e.statusAsSysExitCode());
16.      }
17.      catch (std::exception &e)
18.      {
19.          std::cerr << "std::exception caught by main()."
20.                    << std::endl
21.                    << e.what()
22.                    << std::endl;
23.          exit(1);
24.      }
25.      catch (...)
26.      {
27.          std::cerr << "Non-standard exception in main()."
28.                    << std::endl;
29.          throw;
30.
31.      }
32.      return 0;
33.  }
```

The above is wrapped up by the function `drama::CreateRunDramaTask()`. This function is a template function, with the template argument being a subclass of `drama::Task`. An object of this class will be created with the arguments as passed to `CreateRunDramaTask()` and it `RunDrama()` will method invoked. Any exceptions thrown will be caught and an appropriate report made before the program exits. For example, the above could be replaced by:

```
int main()
{
    drama::CreateRunDramaTask<DramaExampleTask>("EXAMPLE3_3");
    return 0;
}
```

This simple bit of code works in most cases and will be used for all the examples that follow. Note that `CreateRunDramaTask()` is a Variadic template and as a result will work with any sub-class of `drama::task` regardless of the arguments to the constructor.

3.2 Exceptions in action handlers

If an action handler throws an exception that propagates to the caller of `MessageReceived()`, it is caught so as not to allow the DRAMA task to die with bad status and an ERS report sent to the parent task.

If the exception was a `drama::Exception` or sub-class thereof, then the status of the `drama::Exception` will become the completion status of the action and the results of calling `drama::Exception::toString()` are put into the associated ERS report.

If the exception was a `std::exception` of sub-class, then the action completion status is set to `DRAMA2__CPP_STD_EXCEPTION`, and the results of calling `std::exception::what()` is put into the ERS report.

For any other exception, the action completion status is set to `DRAMA2__NON_STD_EXCEPTION`.

This approach is used each time the DRAMA C style interface invoked DRAMA 2 C++ User code.

Example 3.3 (not listed) has three actions, the "`DRAMA`" action throws a `drama::Exception`, the "`STD`" action throws a `std::exception`, whilst the "`OTHER`" action throws something not based on either of the them. Consider trying them. The result is that in each case, the action completes but the task continues to run. An example of running this task is shown below:

Note, the DRAMA 2 and DRAMA 2 Examples error codes are not automatically translated by "`ditscmd`" and other standard tools. You can translate the error code with the "`messana`" program after setting an environment variable to indicate it should use error codes from those sub-systems, see below

```
>> export DRAMA_FACILITIES=./drama2examples_err_msgt.h

>> ./exam3_3&

>> ditscmd EXAMPLE3_3 DRAMA
##DITSCMD_376d:EXAMPLE3_3:DRAMA Exception thrown and reported via ERS
# DITSCMD_376d:EXAMPLE3_3:Example 3.3 throws drama::Exception
# DITSCMD_376d:EXAMPLE3_3:Exception DRAMA Status:0Xc2a8022, "%NONAME-E-
NOMSG, Message Number 0C2A8022"
# DITSCMD_376d:EXAMPLE3_3:Thrown from line 14 in file "sec3/exam3_3.cpp"
DITSCMD_376d:exit status:%NONAME-E-NOMSG, Message Number 0C2A8022
>>
>> messana 0x0C2A8022
Message c2a8022(204111906) - Facility:1066(DRAMA2EXAMPLES), Number:4,
Severity:2
Code:%DRAMA2EXAMPLES-E-EXECEPTION_TEST, Exception test error code

>> ditscmd EXAMPLE3_3 STD

##DITSCMD_38e7:EXAMPLE3_3:std::exception
DITSCMD_38e7:exit status:%NONAME-E-NOMSG, Message Number 0C248022

>> messana 0x0C248022
Message c248022(203718690) - Facility:1060(DRAMA2), Number:4, Severity:2
Code:%DRAMA2-E-CPP_STD_EXCEPTION, C++ RTL Exception generated

>> ditscmd EXAMPLE3_3 OTHER
##DITSCMD_3f49:EXAMPLE3_3:Unexpected exception thrown by Obey - task state
unclear
DITSCMD_3f49:exit status:%NONAME-E-NOMSG, Message Number 0C24802A

>> messana 0x0C24802A
Message c24802a(203718698) - Facility:1060(DRAMA2), Number:5, Severity:2
```

```
Code:%DRAMA2-E-NON_STD_EXCEPTION, Exception which is neither stl nor DRAMA
thrown by handler
```

3.3 More detail on drama::Exception

If required, you can catch a `drama::Exception` and report it to the DRAMA ERS system, flush the report and continue. The `drama::Exception::ErsOut` method will do this, whist `drama::Exception::ErsRep` reports the details to ERS without flushing, leaving you in control.

`drama::Exception` attempts to grab a stack trace at the point where it is invoked. The abilities here are dependent on the operating system support. The data available can be obtained with `drama::Exception::getStackTrace`.

There are various constructors available, but a couple of Macros are also available which wrap up the construction and throw. `DramaTHROW()` takes two arguments, the DRAMA Status code and a string. `DramaTHROW_S()` takes three or more arguments, with the second argument being format string and later arguments being the arguments to that format. The format string contain a "%" character at each point one of the arguments is to be written. Any arbitrary types can be formatted as long as an overload of the << operator is available for that type. This makes it easy to put some contextual information into the exception string, and is type safe in the C++ way.

If you construct a `drama::Exception` object without using the macros, you must ensure you specify the `__FILE__` and `__LINE__` compiler macros to the `file` and `lineNum` arguments to ensure the source location of the exception is known. One case where you might want to do this is where you have called a DRAMA C level routine and want to add the DRAMA ERS information to the exception. This is done using the `AddErs` method, as per the following code snippet:

```
drama::Exception except(true, __FILE__, __LINE__,
                        status,
                        "Failed to get path to task %s",
                        _taskName.c_str());
// Add the ERS information.
except.AddErs();
throw except;
```

You could also construct a `drama::Exception` in this fashion and use the stream << operator to add contextual information to it before throwing it.

3.4 Exceptions within destructors

This is a general C++ issue, rather then anything releated to DRAMA2, but has been seen a few times in DRAMA2 tasks and can cause confusion over what has happened. If an exception is thrown by a destructor, then a program will be aborted – by a call to `std::terminate()`. Any time a program is seen to have aborted via "`terminate()`", this should be considered as a cause.

If a call made from a destructor may throw, you should consider catching the exception if want you program to continue running.

4 SDS and Command Arguments

SDS (the Self Defining data System) is DRAMA's data representation system. SDS consists of an interface (the SDS C Library Routines), an internal representation hidden by the interface, and a documented external representation as a byte-stream. DRAMA uses SDS to transfer data between tasks and allows transfer between machines. SDS transparently deals with issues such as byte order and different floating-point formats in an efficient manner. SDS data consists of named items; each item can refer to a scalar value, an array of scalar values, structured of SDS items or arrays of structures.

Programmers work with SDS items via IDs, so the SDS C Library deals with SDS IDs. Multiple SDS IDs may point to the same data item. A data item may be "internal" to SDS or external (in a byte stream). External items may be stored in files or sent as messages, but some SDS operations are not possible on external items.

When a DRAMA program like "`ditscmd`" sends a message to another task, the arguments to the message are placed in an SDS structure. There is a standard format to these structures, known as the "Arg" format, for the Arg set of C routines used to construct and read them. Arg format SDS Structures are relatively simple but satisfy most uses of SDS within DRAMA programs.

4.1 SDS

The DRAMA2 SDS Interface appears a complex beast, largely due to the number of methods and a few annoying issues that must be dealt with, but most use is simple. The core of the interface is the "`drama::sds::Id`" class. This class wraps up the representation of an SDS ID.

4.1.1 Copying SDS Items

Issues arise when copying SDS Items. If you were to copy an SDS item, do you want to:

1. Copy the underlying SDS data item (possibly many Mega-Bytes in size).
2. Create a new SDS ID referring to the same SDS data item.
3. Create a new C++ object containing the same SDS ID.

And given a choice 2 or 3, what do you want to happen when the item goes out of scope – does the source or destination object free the SDS ID and delete the object?

As a result, copy and assignment of `drama::sds::Id` objects in the normal C++ fashion is prohibited (by deleting the copy constructor and assignment operators). Instead a `Copy()` method is provided for 1 above. The `ShallowCopy()` method implements 3 above whilst 2 is not provided (as 3 covers all required usage).

The move constructor and move assignment operator are provided and are used extensively in the implementation, and hid many of the issues you might otherwise see.

4.1.2 Constructing drama::sds::Id items

Only two standard C++ constructors are provided, the default constructor and the move constructor. All the real work is done with the move constructor, the default constructor only being used to create a target for the move constructor.

To create your SDS item, you use one of a number of static or non-static factory constructors. This approach makes it clearer what you are doing then trying to rely on constructor argument lists to work it out.

The static methods are used when an existing `drama::sds::Id` item is not involved. These are the available static factory constructors:

Method	Description
`drama::sds::Id::CreateFromSdsIdType`	Imports a C SdsIdType ID into a sds::Id.
`drama::sds::Id::CreateByAccess`	Accesses an external SDS item (The SDS Item is in stored in a buffer and will be accessed from the buffer as an external item.).
`drama::sds::Id::CreateByImport`	Imports an external SDS item. (The SDS Item is in stored in a byte buffer and will be imported from the buffer to become an internal item.
`drama::sds::Id::FromFile`	Reads an SDS item from a file.
`drama::sds::Id::CreateTopLevel`	Creates a new top-level item.
`drama::sds::Id::CreateTopLevelArray`	Creates a new top-level array item.
`drama::sds::Id::CreateArgStruct`	Creates a DRAMA Argument style structure. This creates a top-level item, which is a structured SDS, item named “ArgStructure”.
`drama::sds::Id::CreateArgCmdStruct`	Creates a DRAMA Argument style structure and fill it with the supplied values.

For all the above, the destructor will release (free) the underlying SDS ID and delete the SDS data structure (except for those created with `Id::Access`, where no delete is required, and `Id::FromFile` where the buffer the item was read into is instead freed).

The non-static methods are used when another `drama::sds::Id` object is in some way the source. These are the Non-Static Factory Constructor Methods:

Method	Description
`drama::sds::Id::CreateChildItem`	Creates a new child item in an existing SDS item. The object will refer to the child.
`drama::sds::Id::CreateChildArray`	Creates a new child array item in an existing SDS item. The object will refer to the child.
`drama::sds::Id::Cell`	Access a Cell of an array of structures.
`drama::sds::Id::Copy`	Create a copy of the SDS object. The Object will refer to the new top-level internal object that is created.
`drama::sds::Id::Find`	Find an item in a structure by name. The object will refer to the item found.

`drama::sds::Id::Index`	Find an item in a structure by index. The object will refer to the item found

For all the above the underlying SDS ID is released (freed) when the object destructor is invoked. Additionally, for `drama::sds::Id::Copy`, the SDS data structure is deleted by the destructor.

Below we see some examples of the use of the factory constructors as the source for assignment constructors:

```
drama::sds::Id id(drama::sds::Id::CreateTopLevel("Itm", SDS_STRUCT));

drama::sds::Id item1(id.CreateChildItem("ui64", SDS_UI64));
drama::sds::Id item2(id.CreateChildItem("i64",  SDS_I64, extraItem));

std::vector<unsigned long> dims;
dims.push_back(10);
drama::sds::Id item3(id.CreateChildArray("double", SDS_DOUBLE,dims));
```

4.1.3 SDS – Data operations

Example 4–1 is an example of basic SDS operations that we will now work through.

Example 4–1. SDS Usage example

```
1.  #include "drama.hh"
2.  #include "sds_err_msgt.h"
3.
4.  static void Construct(drama::sds::Id *topLevel);
5.  static void ExamineIt(const drama::sds::Id &topLevel);
6.
7.  /*
8.   * Create a printer object used to output an SDS
9.   * structure.
10.  */
11. class MyPrinter : public drama::sds::PrintObjectCR {
12.     virtual void Print(const std::string &) const override;
13. };
14. void MyPrinter::Print(const std::string &line) const {
15.     std::cout << "PP:" << line << std::endl;
16. }
17.
18. int main(int , const char *[])
19. {
20.     /*
21.      * Allow us to translate SDS error codes
22.      *  (Needed as this is not a DRAMA program)
23.      */
24.     MessPutFacility(&MessFac_SDS);
25.
26.     try
27.     {
28.         drama::sds::Id topLevel;
29.         /*
30.          * Construct a structure and check it.
31.          */
32.         Construct(&topLevel);
33.         ExamineIt(topLevel);
34.         /*
35.          * Print via SDS internal printer.
36.          */
```

```
37.          topLevel.List();
38.          /*
39.           * Print via own printer object.
40.           */
41.          topLevel.List(MyPrinter());
42.      }
43.      catch (drama::Exception &e)
44.      {
45.          e.Print();
46.          return 1;
47.      }
48.      return 0;
49.  }
50.
51.  const INT32  MY_INT_VAL = 22;
52.  const double MY_DBL_VAL = 33.33;
53.  const short  MY_ARRAY_SCALAR = 20;
54.  static void Construct(drama::sds::Id *topLevel)
55.  {
56.      drama::sds::Id id(drama::sds::Id::CreateTopLevel(
57.                                    "SdsCppTest", SDS_STRUCT));
58.      /*
59.       * Create a couple of scalar items.
60.       */
61.      drama::sds::Id item1(id.CreateChildItem(
62.                                "myInt", SDS_INT));
63.      drama::sds::Id item2(id.CreateChildItem(
64.                                "myDouble", SDS_DOUBLE));
65.
66.      /*
67.       * Create a 10x20 array of short
68.       */
69.      std::vector<unsigned long> dims(2);
70.      dims[0] = 10;
71.      dims[1] = 20;
72.      drama::sds::Id item3(id.CreateChildArray(
73.                                "myShortArr",
74.                                SDS_SHORT, dims));
75.      /*
76.       * Put the scalar data  items into the structure
77.       *  using low level calls.
78.       */
79.      INT32 myIntVal  = MY_INT_VAL;
80.      double myDblVal = MY_DBL_VAL;
81.      item1.Put(sizeof(myIntVal), &myIntVal);
82.      item2.Put(sizeof(myDblVal), &myDblVal);
83.      /*
84.       * But we could use these simpler higher level
85.       * calls. These do conversions as required.
86.       */
87.      item1.Put((INT32)(MY_INT_VAL));
88.      item2.Put((double)(MY_DBL_VAL));
89.      /*
90.       * Or via name and refer to the top level item.
91.       */
92.      id.Put("item1", (INT32)(MY_INT_VAL));
93.      /*
94.       * Put data into the array item.
95.       */
96.      unsigned long i;
97.      unsigned long count;
```

```
98.      std::vector<unsigned long> arrayDims;
99.
100.     drama::sds::ArrayWriteHelper<short> array;
101.     item3.ArrayAccess(&array, &arrayDims);
102.
103.     count = array.Size();
104.     for (i = 0 ; i < count ; ++i)
105.     {
106.         array[i] = MY_ARRAY_SCALAR*i;
107.     }
108.     /*
109.      * Return the result.
110.      *  Second arg=true says target outlives source
111.      */
112.     topLevel->ShallowCopy(&id, true);
113.
114. }
115. /*
116.  * Examine our structure.
117.  */
118. static void ExamineIt(const drama::sds::Id &topLevel)
119. {
120.     /*
121.      * We find item1 and item2 and then read its value
122.      * using the low-level SDS Get() methods.
123.      */
124.     drama::sds::Id item1(topLevel.Find("myInt"));
125.
126.     INT32 myIntVal;
127.     item1.Get(sizeof(myIntVal), &myIntVal);
128.     if (myIntVal != MY_INT_VAL)
129.     {
130.         DramaTHROW(SDS__TESTERR, "Test error");
131.     }
132.
133.     drama::sds::Id item2(topLevel.Find("myDouble"));
134.     double myDblVal;
135.     item2.Get(sizeof(myDblVal), &myDblVal);
136.     if (myDblVal != MY_DBL_VAL)
137.     {
138.         DramaTHROW_S(SDS__TESTERR, "Test error %, %",
139.                      myDblVal, MY_DBL_VAL);
140.     }
141.
142.     /*
143.      * We could also use these higher level calls
144.      */
145.     item1.Get(&myIntVal);
146.     if (myIntVal != MY_INT_VAL)
147.     {
148.         DramaTHROW(SDS__TESTERR, "Test error");
149.     }
150.
151.     item2.Get(&myDblVal);
152.     if (myDblVal != MY_DBL_VAL)
153.     {
154.         DramaTHROW_S(SDS__TESTERR, "Test error %, %",
155.                      myDblVal, MY_DBL_VAL);
156.     }
157.     /*
158.      * Or via name and reference to the top level item.
```

```
159.       */
160.       topLevel.Get("item1", &myIntVal);
161.       if (myDblVal != MY_DBL_VAL)
162.       {
163.           DramaTHROW(SDS__TESTERR, "Test error");
164.       }
165.       /*
166.        * Now check out array item.
167.        */
168.      drama::sds::Id item3(topLevel.Find("myShortArr"));
169.
170.      unsigned long count;
171.      std::vector<unsigned long> dims(SDS_C_MAXARRAYDIMS);
172.       /*
173.        * To access the array, we use ArrayReadHelper.
174.        *  "dims" will contain the dimensions of the data.
175.        */
176.      drama::sds::ArrayReadHelper<short>  array;
177.      item3.ArrayAccess(&array, &dims);
178.
179.      count = array.Size();
180.      for (int i = 0 ; i < (int)(count) ; ++i)
181.      {
182.          short expected =  MY_ARRAY_SCALAR * i;
183.          if (array[i] != expected)
184.          {
185.              DramaTHROW(SDS__TESTERR,
186.                         "Test error - myShortArr item");
187.          }
188.      }
189. }
```

4.1.3.1 Creating an SDS Structure.

In Example 4–1, creation of the structure is done in the routine `Construct()`. Its argument is the address of an `drama::sds::Id` object which is constructed with the default constructor at line #28. The `Construct()` routine will need to ensure the item constructed is copied into this using `drama::sds::Id ::ShallowCopy()`.

`Construct()` first uses the `drama::sds::Id::CreateTopLevel()` static factory constructor to create the top-level item, at line #56. It then creates a couple of child items; an integer and a double length floating point value. The first argument to each of the `CreateTopLevel()` and `CreateChildItem()` constructors is used to specify the name of the item, the second the SDS type of the item. The third item constructed is a two dimensional array of short integers (SDS type SDS_SHORT), created using the `CreateChildArray()` constructor(). The third argument is a vector with the dimensions. The size of `dims` indicates the number of dimensions (to a maximum of 7).

4.1.3.2 Inserting data into SDS.

When an SDS structure is created, its data items are undefined. Only after you first put data into a structure item is it defined. The low-level SDS `Put()` operation is shown at lines 81 and 82. Here you specify the size of the data item and the address of the data. Any type of data and data of any size can be inserted with this version of `drama::sds::Id::Put()`. You can fill out an entire structure in one operation, which is practical if you have a mapping

between your SDS Structure and a C/C++ "struct". See the SDS Documentation for details on how to arrange this.

But in our example, we have simple scalar items for "item1" and "item2" and there are simpler approaches. There is a simple `Put()` method, with only one argument, which can be used when you have the SDS ID of a scalar (or string) item. This is shown at lines 87 and 88. This method will convert from the type you specify to the type of the item as required, so in this case the code, as it knows the type of the item, has specified that to ensure no conversion is required.

Finally, if you have the SDS ID of the parent item, you can put the value by name, as at line #92. Again, the value is converted if needed and possible to the type of the item. *Additionally, if the item does not yet exist, it is created – so this is a great way of building structures of simple items.*

The array item, `item3`, could have its data put from a C style array of data, using the basic `drama::sds::Id::Put()` method, as per line #81. But often the use of a `drama::sds::ArrayWriteHelper` object is a better approach for arrays of scalar items. These objects provide methods to access the data directly within the SDS item, avoiding otherwise unnecessary copies. They are also the better approach if you need to access only some items in an SDS array. At line #100, an `ArrayWriteHelper` object is created, a sub-class of `ArrayAccessHelper` used for writing to SDS arrays. At line #101 it is supplied to one of the `drama::sds::Id::ArrayAccess()` methods, which will associate it with the SDS data. The "`arrayDims`" item will contain the dimensions of the data, `array.Size()` returns the total number of items. The index operator is used at line #106 to write to the array.

Finally, at line #112, the `ShallowCopy()` method is used to move the underlying SDS ID from "`id`" to "`topLevel`". The second argument to this, set to "true", indicates that the target object (`topLevel`) will outlive the source object (`id`). As a result, the target object becomes responsible for cleaning up the underlying SDS ID and data structure when it goes out of scope.

4.1.3.3 Retrieving data from SDS

The `drama::sds::Id` methods to extract data from SDS are very similar to those used to insert the data, our example does this in its `ExamineIt()` method. At line #127 and again at #135, you can see the use of the low level `drama::sds::Id::Get()` methods. Add per the `Put()` equivalents, these can be used to get entire complex structures out of SDS.

At line #145 and line #151, you see the use the versions of `Get()` which fetch a simple scalar value. And at line #160, you see a version that fetches by name from the top level SDS ID. There are equivalents of these such as `GetLong()`, which return the value directly, but which provide fewer options on the return type (i.e. there is no `GetShort()`).

From line #168 you can see how to access the array item.

4.1.3.4 Other SDS Methods

This section gives a very quick introductions to the other SDS methods which are available:

4.1.3.4.1 Navigating Structures

The main methods for navigating SDS structures are `drama::sds::Id::Find()` and `drama::sds::Id::Index()`. The former finds an item within a Structure by name, the later by Index. The `drama::sds::Id::Exists()` method is an inquiry to determine if a named item exists. There is also `drama::sds::Id::Cell()`, which you will need when working with arrays of SDS Structures. It should be noticed that each element in an array of SDS Structures could be of a different shape/structure.

Methods are also available to determine the details of an item - `drama::sds::Id::GetCode()` will return the SDS Code (type) of an item, `GetDims()` will return the dimensions of array items and `GetName()` will return the name of the item. The `ValidateCode()` method will confirm the item is of a specified SDS type (code) and will throw an exception if it is not. `GetNumItems()` will return the number of items within a structure.

4.1.3.4.2 Navigating structures using iterators

An iterator is provided to allow navigation of structures using the C++11 range construct. The following code segment shows how this may be done:

```
drama::sds::Id myStructId(…);
…
for (auto item:myStructItem)
{
    item.List();
}
```

Here you are trying to operate on each item within the structure represented by "`myStructId`". The range-based for-loop makes this simple, with "`item`" set to each value in turn.

The iterator class being used to implement this is the class `drama::sds::IdIterator`, a forward iterator.

At this point in time, this works with structures and single dimensional arrays of structures. Multi-dimensional arrays of structures are not supported.

4.1.3.4.3 Viewing Structures.

When working with SDS structures, there if often cause to list the structure. Various overloads of the `drama::sds::Id::List()` method are available for this purpose, allowing you to output a listing to `stdout`, a C "`FILE *`", a C++ `std::ostream` or via your own printer object which must implement the `drama::sds::PrintObjectPnt` or `drama::sds::PrintObjectPnt` interface, the `Print()` method of which is invoked for each line of output.

In the same category is the `drama::sds::Id::toString()` method, which converts the contents of an SDS structure to a simple string. Generally, this only works well for small structures.

4.1.3.4.4 Modifying Structures

To add items to an existing structure, you can use `drama::sds::Id::CreateChildItem()` or `CreateChildArray()`. If it is a simple scalar item, you can also use overload of `drama::sds::Id::Put()` with a name

argument, as this will create the item if it does not exist. An array of structures can be filled in with copies of an existing SDS structures using `FillArray()`.

You can add a top-level item into an existing structure using `drama::sds::Id::Insert()`, or the reverse - extract a sub-structure into its own top-level structure using `Extract()`. You can delete an existing item using `Delete()`.

You can rename an item using `drama::sds::Id::Rename()` and resize SDS Arrays with `Resize()`.

4.1.3.4.5 Extra Data

Each SDS item may have an item of "Extra Data" associated with it. This has rarely been used in practice, but is sometimes of use. The raw "Extra Data" can be any byte array, but in practice, it is always treated as a string and this interface presumes that.

Each of the constructors that create an SDS item provides an optional "extra" argument. If not an empty string, it will be associated with the item. There are also methods to retrieve this item - `drama::sds::Id::GetExtra()` and set it - `PutExtra()`. Extra data is listed in `List()` operations, but otherwise must be fetched by the application to be used.

4.1.3.4.6 Export/Import of structures

SDS Structures are created in an efficient internal memory structure. But they can be "exported" to a byte stream and then re-imported from a byte stream. This allows them to be stored in files and sent in messages. The "external" structure layout if defined in the SDS documentation and in theory alterative readers can be created.

The `drama::sds::Id::Export()` method will export into a buffer. But be careful that if SDS data is not yet "defined", then it exported that way. Undefined data gives an error if you try to read the data. To be sure that any undefined data is defined, use `ExportDefined()`. To get the size of the buffer needed, invoke `Size()` or `SizeDefined()`. The Export methods do not attempt to resize the buffers (since that limits the types that can be specified as the buffer).

External data can be accessed from within the external buffer using `drama::sds::Id::CreateByAccess()` or imported, creating an internal SDS item, using `CreateByImport()`. You should import the structure if you wish to change the layout of the structure. You can change the values of data in an external SDS item, but you can't change the layout.

The `drama::sds::Id::Write()` method is a utility that exports an SDS item and writes it straight to a specified file. The `ReadFromFile()` static factor constructor will likewise read from a file and access the item.

4.1.3.4.7 Export/Import of IDs

The `drama::sds::Id` class is a wrapper around the C level SDS ID, itself a reference to the actual SDS data structures. A method exists which allow C style SDS ID's to be imported, the static factory constructor `drama::sds::Id:CreateFromSdsIdType()`. The optical arguments indicate what the destructor should do with the underlying SDS ID, if you want the SDS ID free-ed, deleted or, if it was read from a file, the buffer free-ed with `SdsReadFree()`. The defaults presume you want none of these done, which is the best choice in most cases where say a C

SDS ID has been passed to your method, is used within your method and is tidied up by the caller.

It is also possible to export from `drama::sds::Id` to C style SDS ID's. There is an operator that will convert to `SdsIdType`, which is useful in calling C interfaces that access your SDS Item but won't want to delete the item or free the ID. More control is available with the `COut()` method, which allows you to find out what was supposed to happen when this object is destroyed and to tell the object not to do those operations (E.g. if for example the C code is going to call `SdsDelete()` and `SdsFreeId()` on the ID returned by `COut()`).

4.1.3.4.8 Copying

The `drama::sds::Id::Copy()` factory constructor will return a deep copy of the source object. By a deep copy, we mean that the entire SDS structure including all of its data is copied into a new internal item that can be accessed and changed dependently of the original item. In addition to use this if you just want to copy an item, you will need to use it if your source object is an external item (method `IsExternal()` can tell you that) and you wish to modify the structure of the item.

The alternative copy is the "Shallow Copy". A Shallow Copy allows a `drama::sds::Id` item to take control of an SDS ID from somewhere else. The `drama::sds::Id::CreateFromSdsIdType()` method mentioned in the previous section, is, in effect, a shallow copy factory constructor where the source is a C style SDS ID. There are also two `drama::sds::Id:ShallowCopy()` methods. Both of these give a way for an `drama::sds::Id` object to take control of another SDS ID, as per line #112 of Example 4–1, where we need to pass an item we have constructed back to caller. For one of these, the source is another `drama::sds::Id` item. In this case, we must indicate if the target item will outlive the source item. If it will, then the target must take over full control of the underlying SDS ID and delete the structure and free the ID when it goes out of scope, rather then when the source goes out of scope. For the other ShallowCopy() method, the source s a C style SDS ID and the other arguments must indicate if the ID is to be freed, deleted and/or read-freed when it goes out of scope. (Note, you set the `delete` flag items which have been created, you set the `readfree` flag for items which have been read via `SdsRead()` or similar.

4.1.3.4.9 Direct access to the SDS Data.

The Get/Put methods are one way to work with SDS data, but they all involve copying the data. This may introduce performance problems if you are accessing the data a lot or say have a large array that you want to access only occasional items of. SDS Provides a technique to allow direct access to the data as stored by SDS. The `drama::sds::Id::Pointer()` method provides the lower level interface to this. Note that having used this, if you update that data you should invoke `drama::sds::Id::Flush()` to ensure any update is written out[3].

But `drama::sds::Id::Pointer()` is very much a "C-style" approach to the problem. A better interface is provided by the `drama::sds::DataPointer` class. This class is a

[3] `drama::sds::Id::Flush()` is a null operation on all currently supported implementations of SDS, but is retained for compatibility reasons.

subclasses of `std::unique_ptr<>` and provides a safer way of accessing the data. There is also an array specialization of this class. The template types here can be any C++ POD[4] (Plane Old Data) types, such as SDS Scalar types or C structures containing SDS Scalar types.

Example 4–2 is a rewritten versions of `ExamineIt()` from Example 4–1, using `drama::sds::DataPointer`. This example only reads the data items, it is possible write them as well if the pointers are not defined as "`const`".

Example 4–2. Accessing SDS data via pointers

```
1.   static void ExamineIt(const drama::sds::Id &topLevel)
2.   {
3.       /*
4.        * Find the integer item and then access its value
5.        *   via pointer.
6.        */
7.        drama::sds::Id item1(topLevel.Find("myInt"));
8.        const drama::sds::DataPointer<INT32> myIntValPnt(item1);
9.        if (*myIntValPnt != MY_INT_VAL)
10.       {
11.           DramaTHROW(SDS__TESTERR, "Test error");
12.       }
13.      /*
14.       * Find the double item and then access its value
15.       *  via pointer.
16.       */
17.       drama::sds::Id item2(topLevel.Find("myDouble"));
18.       const drama::sds::DataPointer<double> myDblValPnt(item2);
19.       if (*myDblValPnt != MY_DBL_VAL)
20.       {
21.           DramaTHROW_S(SDS__TESTERR, "Test error %, %",
22.                        *myDblValPnt, MY_DBL_VAL);
23.       }
24.       /*
25.        * Find the array of short and check it out.
26.        */
27.      drama::sds::Id item3(topLevel.Find("myShortArr"));
28.      const drama::sds::DataPointer<short[]> array(item3);
29.
30.      int i = 0;
31.      for (auto it = array.cbegin();
32.           it != array.cend() ;
33.           ++it, ++i)
34.      {
35.          short expected =  MY_ARRAY_SCALAR * i;
36.          if (array[i] != expected)
37.          {
38.              DramaTHROW(SDS__TESTERR,
39.                         "Test error - myShortArr item");
40.          }
41.      }
42.      /*
43.       * Note - nothing stops you updating the values
```

[4] C++11 defines PODs as trivially copyable types, trivial types, and standard-layout types. POD is defined recursively If all your members and bases are PODs, you're a POD. A POD has: No virtual functions, No virtual bases, No references, No multiple access specifiers. The most important aspect of C++11 PODs is that adding or subtracting constructors do not affect layout or performance. Static assertions are used by these DRAMA2 methods to ensure template type arguments are POD types

```
44.       *  using these approaches, with the non-const
45.       *  pointers
46.       */
47.  }
```

4.1.3.4.10 SDS Compiler.

Whilst not a feature of `drama::sds`, it should be noted that SDS provides a way of quickly moving data to/from C struct's (C++ POD types). The SDS Compiler will take a C include file with a struct definition and creates a file containing a C function. When this C function is invoked, it generates an SDS structure equivalent to the C struct. You can then use the Get/Put methods to move data between the C and SDS equivalents in one operation. Please see the SDS Manual for more information.

4.2 DRAMA Argument Structures

Messages sent between DRAMA tasks can contain an SDS structure argument of any complexity, with the size limited only by the size of the message buffers set up when communications between the two tasks are initiated.

But, to allow programs to be sent messages from utility programs such as "ditscmd", some standards are required. To accept arguments from utility User Interfaces, a DRAMA Action should accept an SDS Structure containing items named "Argument1" through to "Argument<n>". Typically, these items are string or scalar values. These are to be converted as required to the type needed by the client program. "`ditscmd`" will normally send a structure of all string arguments, but clients that need integer or floating point values should convert them as required. The top-level SDS structure is normally named "ArgStructure", but in most cases this name is ignored (one `gitarg` namespace case does use the name, see below).

The `drama::sds::Id` class provides several static factory constructors to help in creating such structures. `drama::sds::Id::CreateArgStruct()` creates a top-level SDS item ready for inserting values. Normally you would use the `Put()` with name methods to insert each value, but the item is a standard SDS Structure, so use of other SDS methods is possible. There is also the `drama::sds::Id::CreateArgStructCmd()` constructor. It creates an argument structure and fills out the items "Argument1" through to "Argument<n>" using values in container passed to the method. If you already have such a structure and want to add items to it, you could also use the `AddToArgStructCmd()` method, which is similar to the `CreateArgStructCmd()` constructor.

Example 4–3 shows an example of using these methods. The structure is created at line #9, containing three string items from a vector. Line #14 shows some short values being added, whist line #20 shows some stings representing Boolean values being added. There are also some examples of using the `Put()` with name methods, at line #25 and line #28.

Example 4–3. Constructing Arg style SDS structures

```
1.   static void Construct(drama::sds::Id *topLevel)
2.   {
3.       /* A Vector of string items */
4.       std::vector<std::string> strArray{
5.           "StrItem1", "StrItem2", "StrItem3" };
6.
7.       /* Create ArgStructure SDS item, one item for
8.          each in strArray  */
```

```
9.      drama::sds::Id id(
10.         drama::sds::Id::CreateArgCmdStruct(strArray));
11.
12.     /* Add an array of short values to the structure */
13.     short shortArray[]= { 4, 5, 6, 7 };
14.     id.AddToArgCmdStruct(shortArray, strArray.size()+1);
15.
16.     /* Add an array of strings to be treated
17.        as bools when read */
18.     std::vector<std::string> boolArray{
19.         "yes", "NO", "TRUE", "FALSE" };
20.     id.AddToArgCmdStruct(
21.         boolArray,
22.         strArray.size()+drama::ArraySize(shortArray)+1);
23.
24.     /* Add a string as a named item. */
25.     id.Put("BY_NUM", "String by Num");
26.
27.     /* Add a named item, referring to a file */
28.     id.Put("FILE", "./raw_struct.sds");
29.
30.     /*
31.      * Return the result.
32.      *  Second arg=true says target outlives source
33.      */
34.     topLevel->ShallowCopy(&id, true);
35.
36.
37. }
```

The receiver of such structures could use the drama::sds::Id::Get() by name series methods to retrieve the data, but classes in the `gitarg` namespace provides more options, as shown in the next section.

4.3 The `gitarg` namespace

Programs accepting arguments from the user or other programs typically need to validate those values in some way. They may also want to apply defaults if a value is not given. Strings may converted to enumerated types, including Boolean. The `gitarg` namespace provides classes that can be used by a DRAMA program to interpret SDS Argument structures using such rules.

The table below summarizes the types provided.

Class	Description	Comments
`drama::gitarg::Bool`	Implements a Boolean type that can be initialized from a value in an SDS structure.	User can supply various string representations of TRUE and FALSE, e.g. YES and NO, ON and OFF etc, as required
`drama::gitarg::Enum`	Implements an Enumerate type that can be initialized from a value in an SDS structure.	A template class, user provides two types to the template, one the Enum value they want to use, the other a helper class they must implement
`drama::gitarg::Int`	Implements an Integer	A range validated integer

Class	Description	Comments
	type that can be initialized from a value in an SDS structure.	represented using “long int”.
`drama::gitarg::Real`	Implements a Real number type that can be initialized from a value in an SDS structure.	A range validate Real number represented using “double”.
`drama::gitarg::String`	Implements an overload of `std::string` which can be initialized from a value in an SDS structure.	Can be used as a `std::string` type, but allows initialization from a value in an SDS structure.
`drama::gitarg::Id`	Implements an overload of `drama::sds::Id` which can be initialized from a value in an SDS structure.	Allows complex structures or references to SDS files to be passed as arguments.

Example 4–4 below (the source is part of the file containing Example 4–3) shows various examples of using these classes. Each class implements a Constructor and a `Get()` method, both of which take as an argument an SDS structure. An important point to note is that arguments are fetched by Name if possible and then by position. The idea is that you can support well defined argument names, which you might get if writing the task that sends the message as part of a system of tasks, but you can also support arguments sent by general DRAMA commands such as “`ditscmd`”, which don’t (by default) allow you to specify the names for each argument. If your argument is not found by the name specified, these methods look in the specified position. Be warned that this can go astray if you have a mixture (which our example does), but in real world examples, it achieves the desired result.

In the example, we are extracting information from the structure constructed in Example 4–3. The `drama::gitarg::Enum` type requires that you have set up the Lookup argument and the Enum argument to the template in an appropriate way, as shown from lines 1 to 28. Once that is done, the constructor at line #43 will read the argument and construct the Enum from the argument named “Argument1” or at position number 1 in the SDS structure, if one named “Argument1” does not exist. If the supplied value does not match one of the expected values, an exception is thrown.

Further in the example, construction of an integer from an argument that must be in the range -10 to 100 (line #58), Boolean examples (lines 73 to 76), a string example (line 90) and finally the more complicated `drama::gitarg::Id` class from line #102.

Example 4–4. Reading a structure with gitarg (for source - see example 4.3 source)

```
1.   /*
2.    * We want to accept an argument which translates to one of the
3.    * Enum values in MyArgEnum.  We must accept strings, which
4.    * gitarg::Enum can convert to the enumerated values.
5.    *
6.    * The enumerated values must have integer values from 0 up.
7.    * There needs to be an extra "Invalid" value, which is
8.    * typically named "Invalid" but anything is allowed.
9.    */
10.  enum MyArgEnum { Item1=0, Item2, Item3, Invalid };
11.  /*
12.   * This is a sub-class of EnumLookupClass used to convert
```

```
13.   * between the strings representing MyArgEnum and the
14.   * integer value equivalents of the enum values.
15.   */
16.  class MyArgEnumLookup : drama::gitarg::EnumLookupClass {
17.  public:
18.      /* Returns the maximum normal value of the enum as int */
19.      unsigned int GetMaxValue() const override final {
20.          return ((int)(Invalid))-1;
21.      }
22.      /* Return a pointer to an array of string equivalents */
23.      const char  ** GetStringArray() const override final {
24.          static const char *table[] =
25.              { "STRITEM1", "STRITEM2", "STRITEM3", 0 };
26.          return table;
27.      }
28.  };
29.
30.  static void ExamineIt(const drama::sds::Id &topLevel)
31.  {
32.      /* We could use SDS Arg style calls to read the structure
33.       * but instead we will use gitarg, it has a lot of useful
34.       * features.
35.       */
36.
37.      /*
38.       * This value is fetched from the item named "Argument1"
39.       * (constructor arg #2) within the SDS structure in
40.       * topLevel, or the 1st (constructor arg #3) item in the
41.       * structure if the item named "Argument1" does not exist.
42.       */
43.      drama::gitarg::Enum<MyArgEnumLookup, MyArgEnum> arg1(
44.          topLevel,      // Source SDS structure
45.          "Argument1",   // Name of item we are interested in
46.          1              // Position of item we are interested in
47.          );
48.
49.      std::cout << arg1 << " (" << std::string(arg1)
50.                << ")"  << std::endl;
51.
52.      /*
53.       * Declare an item, which is of type gitarg::Int, an
54.       * integer within a specified range.  Fetch from the
55.       * item named "Argument4", or the 4th item in the
56.       * structure if that does not exist.
57.       */
58.      drama::gitarg::Int<-10, 1000> arg4(
59.          topLevel,      // Source SDS structure
60.          "Argument4",   // Name of item we are interested in
61.          4              // Position of item we are interested in
62.          );
63.
64.      std::cout << "Argument 4 has the value:"
65.                << arg4 << std::endl;
66.
67.      /*
68.       * Try our set of Boolean values.
69.       *
70.       *  Please ignore the ArgCvt() errors at this point - they
71.       *    won't appear in a DRAMA program.
72.       */
73.      drama::gitarg::Bool arg8 (topLevel, "Argument8", 8);
```

```
74.      drama::gitarg::Bool arg9 (topLevel, "Argument9", 9);
75.      drama::gitarg::Bool arg10(topLevel, "Argument10", 10);
76.      drama::gitarg::Bool arg11(topLevel, "Argument11", 11);
77.
78.      std::cout << "Arguments 8 through 10: "
79.                << arg8 << " "
80.                << arg9 << " "
81.                << arg10 << " "
82.                << arg11 << " "
83.                << std::endl;
84.
85.      /*
86.       * Get argument 12 value as a string. Also demonstrating
87.       * fetching by number rather the name, since the name
88.       * "Argument12" does not exist.
89.       */
90.      drama::gitarg::String arg12_s(topLevel, "Argument12", 12);
91.      std::cout << "Argument 12 has the value\""
92.                << arg12_s
93.                << "\""
94.                << std::endl;
95.
96.      /*
97.       * The 19th' argument (if we don't find one named "FILE",
98.       * which we will) refers to an SDS structure. That could
99.       * be passed in the argument, but in this case, the name
100.      * of a file containing the structure is passed.
101.      */
102.     drama::gitarg::Id arg19(topLevel, "FILE", 19);
103.     arg19.List();
104.
105.     /*
106.      * Same argument, via Get() method and by position since
107.      * we don't have an argument named Argument13.
108.      */
109.     drama::gitarg::Id arg13;
110.     arg13.Get(topLevel, "Argument13", 13);
111.     arg13.List();
112. }
```

The `drama::gitarg::Id` class allows more complicated SDS structures to be passed as part of a command argument structure. Remember that SDS structures can be of any complexity. With the `drama::gitarg::Id` class, a complicated structure can be passed directly in the supplied structure, or via an SDS file the name of which is specified.

4.4 Accessing action arguments

The `GetArgument()` method of `drama::MessageHandler` will return any argument to the action. You should check that you have been supplied a value before you use this, otherwise the SDS calls will throw an exception.

Example 4–5 show an example a `MessageReceived()` implementation which does this (at line #4). The `bool` operator on the SDS object (line #5) allows you to check if an argument has been supplied. This example just lists the contents, but you can of course use any of the features shown in the previous sections to access the argument values. But to remember that the underlying SDS structure is deleted when `MessageReceived()` returns. *If you need to keep it about, `SdsID::Copy` it!*

Example 4–5. Accessing message arguments

```
1.      drama::Request MessageReceived() override {
2.
3.          // Access action argument
4.          drama::sds::Id arg = GetEntry().Argument();
5.          // And list it.
6.          if (arg)
7.              arg.List();
8.          return  drama::RequestCode::End;
9.      }
```

4.5 Returning SDS structures to the action's source talk

An action may also return SDS structures to the task that initiated the action. There are two ways of doing this – trigger messages and action completion message arguments.

First we need to understand a bit of terminology. DRAMA normally talks about "Parent Actions" and "Child Actions". The "Parent Action" is the DRAMA action in (normally) another task that originated the message to start an action. The "Child Action" is the action that was started as a result of the message. So far, we have been dealing only with the child. So when action X in task A sends an Obey message to start action Y in task B, X is the parent action, Y is the child action.

4.5.1 Trigger Messages

Trigger messages allow a child action to send some information to their parent action, but the child can continue to work. The child action can send multiple trigger messages as required. What the parent does with the trigger messages is determined by the interface negotiated between the authors of the two tasks.

Sending trigger messages is very simple, using the `SendTrigger()` method of `drama::MessageHandler`. The argument is the SDS structure to send. The message is sent immediately and you can send multiple triggers. Example 4–6 shows an example of doing this. When you try this example with "`ditscmd`", a simple message is output indicting the trigger has been received, with the SDS value converted to a string.

Example 4–6. Sending Trigger messages

```
1.  class HelloAction : public drama::MessageHandler {
2.  public:
3.      HelloAction() {}
4.      ~HelloAction() {}
5.  private:
6.
7.      drama::Request MessageReceived() override {
8.
9.          drama::sds::Id trigArg(
10.             drama::sds::Id::CreateArgStruct());
11.         trigArg.Put("Argument1", "Hi there");
12.         trigArg.Put("Argument2", "Quick brown fox");
13.         SendTrigger(trigArg);
14.
15.         return  drama::RequestCode::End;
16.     }
17. };
```

4.5.2 Action Completion Message Arguments

Whilst trigger messages are a useful feature, a more common way to return an SDS structure to the parent action is with the message that indicates the child action has completed. This is a better match in many cases – you start an action, which does something and returns its result. The `SetReturnArg()` methods of `drama::MessageHandler` are used to do this. Note that such arguments are only sent if your action returns with a request code of `RequestCode::End` or `RequestCode::Exit`.

There is a complexity with return (or output) arguments. Your action code will have completed (and the destructors in your action code have been run) before DRAMA sends the message. As a result, any SDS structure you supplied must outlive the completion of your action.

If you call `SetReturnArg()` with the `drama::sds::Id` argument passed by value (actually by const-reference) then (by default) a deep copy of the SDS item is made. A reference to this copy will be stored by DRAMA and the copy deleted when DRAMA is finished with it. This approach works ok with small SDS items, or where you need to keep the original about but are going to modify it. But other approaches should be considered for large items. A second argument can tell DRAMA not to copy the item, but instead just keep a reference to the underlying SDS ID. You might do this if the SDS structure item is static in some way, such that it will outlive the completion of your action.

The other way to call `SetReturnArg()` is with the `drama::sds::Id` argument passed by (non-const) pointer. If you do this, then DRAMA takes over control of tidying up the underlying SDS item –deleting it when it is done with it. The reason a non-const pointer is required is so that your `drama::sds::Id` item can be modified to ensure its destructor does not tidy it up. This approach is probably the best in most cases and is what is demonstrated by Example 4–7 below.

Example 4–7. Setting a completion message argument

```
1.   class HelloAction : public drama::MessageHandler {
2.   public:
3.       HelloAction() {}
4.       ~HelloAction() {}
5.   private:
6.
7.       drama::Request MessageReceived() override {
8.           drama::sds::Id outArg(
9.               drama::sds::Id::CreateArgStruct());
10.          outArg.Put("Argument1", "Hi there");
11.          outArg.Put("Argument2", "Quick brown fox");
12.
13.          // Set the output argument. DRAMA Takes control
14.          SetReturnArg(&outArg);
15.
16.          return  drama::RequestCode::End;
17.      }
18.  };
```

When you try Example 4–7 with "`ditscmd`", the reply argument is simply converted to text and output. You could also specify the "`-v`" option to "`ditscmd`" to cause the structure to be listed in detail.

4.6 Complex parameters

In section 2.2 we examined adding and working with task parameters. But all the examples in that section were simple parameters, either scalar items or strings. In reality, parameters can be anything that SDS can represent. But in the complex cases, you have to manage more of the details yourself and only now do you have information to understand how to do that. Both the `drama::Parameter` and the `drama::ParSys` approaches support complex parameters.

4.6.1 Complex parameters with drama::Parameter

`drama::Parameter` provides an explicit template specialization for the type "`drama::sds::Id`", which is instantiated at line #21 in Example 4–8. The constructor (invoked at line #27) requires that you supply an SDS ID via an `sds::IdPtr`, containing your complex structure.

In this example, a static method is used to create the structure – line number #6. The `drama::sds::IdPtr` type is a instantiation of `std::shared_ptr<>`, with `drama::sds::Id` as the template argument, and is used where we might want pointed to `drama::sds::Id` items, as in this case. The method is static because it needs no access to the class but is called by the constructor.

The `drama::Parameter< drama::sds::Id>` instantiation provides `Get()` and `Set()` methods to get and set the value of the parameter, see line #37.

Example 4–8. Complex parameters with the drama::Parameter class

```
1.   class DramaExampleTask : public drama::Task {
2.   private:
3.       HelloAction HelloActionObj;
4.
5.       // Create the structured parameter.
6.       static drama::sds::IdPtr CreateMyStructParam() {
7.           drama::sds::IdPtr param(
8.               std::make_shared<drama::sds::Id>(
9.                   drama::sds::Id::CreateTopLevel("PARAM",
10.                                                  SDS_STRUCT)));
11.          param->Put("STRUCT_VALUE_1", 11);
12.          param->Put("STRUCT_VALUE_2", 21);
13.          return param;
14.      }
15.
16.
17.  public:
18.      /*
19.       * Structured parameter declaration..
20.       */
21.      drama::Parameter<drama::sds::Id> structParam;
22.      /*
23.       * Constructor.
24.       */
25.      DramaExampleTask(const std::string &taskName) :
26.          drama::Task(taskName),
27.          structParam(TaskPtr(), "STRUCT_PARAM",
28.                      CreateMyStructParam()) {
29.
30.          Add("HELLO",
31.              drama::MessageHandlerPtr(
32.                  &HelloActionObj,
33.                  drama::nodel()));
```

```
34.          // Standard simple EXIT action.
35.          Add("EXIT", &drama::SimpleExitAction);
36.
37.          drama::sds::Id theParam = structParam.Get();
38.          theParam.List();
39.      }
40.  };
```

4.6.2 Complex parameters with drama::ParId

The approach used by `drama::ParSys` is a little different. A separate class is provided, `drama::ParId`. This class is a subclass of `drama::sds::Id`, but only provides the one constructor, which allows it to be constructed to refer to a named parameter. One additional method (compared to `sds::Id`) is provided – `Update()`. This method is used to notify DRAMA that you have changed the contents of the item, and is required to ensure that any tasks, which are "monitoring" the value of the parameter, are notified of the change.

A number of the `drama::sds::Id` methods have the potential to destroy the parameter, so these are made "private" to help avoid misuse, and then overridden so that if they are called (via the drama::sds::Id interface), an exception is thrown. `SetFree()`, `SetDelete()`, `ClearDelete()`, `Outlive()`, `Delete()`, `Extract()`, `Rename()`, `COut()`, and `ShallowCopy()` are all impacted here.

Otherwise, you can use `drama::sds::Id` methods as required to update the parameter value. Example 4–9 below shows an example of an action doing this after a the parameter was constructed in a similar way to Example 4–9. It constructs a `drama::ParId` object to access the parameter at line #8. It then does some work with it, listing it, changing a value (with a "named" `sds::Id::Put`), telling DRAMA it has updated it and then lists it again.

Example 4–9. Complex parameters with drama::ParId.

```
1.   class HelloAction : public drama::MessageHandler {
2.   public:
3.       HelloAction() {}
4.       ~HelloAction() {}
5.   private:
6.
7.       drama::Request MessageReceived() override {
8.           drama::ParId structId(GetTask(), "STRUCT_PARAM");
9.           structId.List();
10.          structId.Put("STRUCT_VALUE_2", 23);
11.          structId.Update();
12.          structId.List();
13.          return  drama::RequestCode::End;
14.      }
15.  };
```

5 Action Rescheduling

DRAMA has always implemented a form of "Cooperative Multitasking" of actions. In "Cooperative Multitasking", there is no forced rescheduling of "threads" by the operating system; instead user code must behavior in an appropriate way to ensure rescheduling is possible. When DRAMA was originally implemented, this was the only highly portable way to implement any type of Multitasking within a single program.

This design requires that actions in a task should "Reschedule" to wait for events, rather then to block waiting for them. When an action "Reschedules", the DRAMA main loop will process the next message it has (or will) received possibly triggering another action to run or the same action to be "Rescheduled". When this is done right, tasks are very responsive to messages and points for cancelling or aborting actions are well defined. The approach works well when actions are sending/receiving messages or are working with hardware. It is harder to get right if actions have CPU intensive work to do, but DRAMA does provide help for many of those cases.

DRAMA2 does allow you to implement actions using modern threads (see section 6), avoiding the need for this "Rescheduling" approach in many cases, but the "Rescheduling" approach is potentially useful in some cases and is potentially more efficient, and since the implementation was needed to implement threaded actions, it has been made public and we describe it in this section.

5.1 Basic rescheduling.

The `GetEntry()` method of `drama::MessageHandler` provides information about why an action entry occurred. One item it provides is the "Sequence" count, indicating what reschedule event is currently being run. `GetEntry().Sequence()` will return zero when the action is first invoked and will be increment for each reschedule event until the action completes. It will be reset to zero each time the action is invoked (i.e. each time a is invoked by a message sent to the task).

Various other bits of information is available from `GetEntry()`. Of particular interest is `GetEntry().Reason()`, which returns a code indicating what triggered the entry. This method will but used in later examples.

The return value from the `MessageReceived()` method tells the DRAMA main loop what should be done. We have previously seen the use of the value "`drama::RequestCode::Exit`", which causes the task to exit, and the "`drama::RequestCode::End`", which causes the action to complete. There are various other possibilities; `Stage`, `Wait`, `Message` and `Sleep`.

Example 5–1 shows the simplistic example of rescheduling. This action implementation checks the Action sequence count (line #7) and if it is zero, requests a "Stage" reschedule event (line #11). A "Stage" reschedule event causes the action to be rescheduled immediately, after allowing for processing of any messages queued for the task. It is used to break up operations to keep the task responsive and, in some cases, helps in structuring the task. When invoked on any other sequence count value, it requests the action end (line #17).

Example 5–1. Basic Rescheduling

```
1.   class HelloAction : public drama::MessageHandler {
2.   public:
3.       HelloAction() {}
```

```
4.      ~HelloAction() {}
5.  private:
6.      drama::Request MessageReceived() override {
7.          if (GetEntry().Sequence() == 0)
8.          {
9.              MessageUser(
10.                 "Hello World - from DRAMA 2, first entry");
11.             return  drama::RequestCode::Stage;
12.         }
13.         else
14.         {
15.             MessageUser(
16.                 "Hello World - from DRAMA 2, second entry");
17.             return  drama::RequestCode::End;
18.         }
19.     }
20. };
```

Below we see the results of running this example

```
>> ./exam5_1&
>> ditscmd EXAMPLE5_1 HELLO
DITSCMD_dc4:EXAMPLE5_1:Hello World - from DRAMA 2, first entry
DITSCMD_dc4:EXAMPLE5_1:Hello World - from DRAMA 2, second entry
```

5.2 Rescheduling after a delay.

Whist the "`Stage`" reschedule event is useful, a more common reschedule request is to delay your action. First is should be noted that the return type of `MessageReceived()` is `drama::Request`. We have so far been returning a `drama::RequestCode` enumerated value, which can be used to construct the required `drama::Request` type. To reschedule your action with a delay, you need to return both the `RequestCode` of `Wait` and the time (floating point seconds) you want to delay your action.

Example 5–2 is a minor change to Example 5–1 which implements a 10 second wait.

Example 5–2. Rescheduling after a delay

```
1.  class HelloAction : public drama::MessageHandler {
2.  public:
3.      HelloAction() {}
4.      ~HelloAction() {}
5.  private:
6.      drama::Request MessageReceived() override {
7.          if (GetEntry().Sequence() == 0)
8.          {
9.              MessageUser(
10.                 "Hello World - from DRAMA 2, first entry");
11.             return  drama::Request(
12.                 drama::RequestCode::Wait, 10) ;
13.
14.         }
15.         else
16.         {
17.             MessageUser(
18.                 "Hello World - from DRAMA 2, second entry");
19.             return  drama::RequestCode::End;
20.         }
21.     }
```

```
22.  };
```

5.3 Other reschedule reason codes.

The reschedule code of "`Message`" is traditionally used when you are rescheduling to wait for replies to messages sent to other tasks. The "`Sleep`" code is traditionally used to wait for other events, such as hardware events were a signal handler might be triggered. It could wake up the action using one of the (DRAMA C) `DitsSignal()` series of functions. Both `Message` and `Sleep` codes would normally have a timeout associated with them, so they are returned in a similar fashion to `Wait`, above. But unlike `Wait`, the timeout is not compulsory.

There is currently no implementation difference between the `Message` and `Sleep` codes, and the only extra feature with the `Wait` code is that the timeout is compulsory. It was originally thought that DRAMA might implement different checks based on the code used, but this was never done.

The `Message` and `Sleep` codes are rarely used in DRAMA2, as threads provides alternatives much of what they are needed for.

The `drama::Request` class also allows you to return an action completion status codes, via a different constructor, but this is rarely done in DRAMA2.

5.4 Changing the Action Handler

When breaking an action up with Rescheduling, you often naturally want different functions/object methods involved in implementing each reschedule event. A naïve way to implement this would be to check the action sequence number and invoke a different function or object method based on the sequence. But that is error prone and makes it more difficult to design library code for use in task implementations. DRAMA provides an alterative; which is to change the "Handler" to be used for the next reschedule event of the action. The handler is reset to the original value when the action completes such that the same handler always handles the start of the action.

The `PutObeyHandler()` method of `drama::MessageHandler` provides a way of specifying an alterative object, implementing the `drama::MessageHandler` interface, to be used for the next reschedule event.

Example 5–3 is a reimplementation of Example 5–1 using this approach. A member object of class `MyObeyRescheduleHandler` is used to handle the reschedule event. The `PutObeyHandler()` method is invoked at line #24 to enable its use for the next reschedule event. Note that the object involved does not need to be a member of the `HelloAction` class, but in this case that is the most convenient structure.

Example 5–3. Changing Obey handlers

```
1.   class MyObeyRescheduleHandler : public drama::MessageHandler {
2.   public:
3.       MyObeyRescheduleHandler() {}
4.       ~MyObeyRescheduleHandler() {}
5.       drama::Request MessageReceived() override {
6.           MessageUser(
7.               "Hello World - from DRAMA 2, second entry");
8.           return  drama::RequestCode::End;
9.       }
10.  };
```

```
11.
12.  // Action definition.
13.  class HelloAction : public drama::MessageHandler {
14.  public:
15.      HelloAction() {}
16.      ~HelloAction() {}
17.  private:
18.
19.      MyObeyRescheduleHandler _reschedHand;
20.
21.      drama::Request MessageReceived() override {
22.          MessageUser(
23.              "Hello World - from DRAMA 2, first entry");
24.          PutObeyHandler(
25.              drama::MessageHandlerPtr(&_reschedHand,
26.                                       drama::nodel()));
27.          return  drama::RequestCode::Stage;
28.      }
29.  };
```

5.5 Handling Kick Messages when Rescheduling.

An action in a DRAMA task may be “Kicked”. This means another DRAMA task has sent that action a “Kick” message. Kick messages can be sent by “ditscmd” by adding the “-k” option, but otherwise look the same as an “Obey” message. Only a running action may be kicked, if the action is not running the kick message will be rejected.

A Kick message is often be used to abort an action, but it can also be used to provide additional information to the action. To process a Kick message, the action must have indicated to DRAMA it is interested in receiving Kick messages, and it must rescheduled to receive them.

In DRAMA 2, an action may enable reception of Kick messages by specifying a handler for them using the The `PutKickHandler()` method of `drama::MessageHandler`. The specified object must implement the `drama::MessageHandler` interface.

Example 5–4 is a modification to Example 5–3 that provides a delay before the reschedule event happens and implements handling of kick messages, via the `MyKickHandler` class at line #15. This is specified as the handler at line #44

Example 5–4. Kick Handlers

```
1.   class MyObeyRescheduleHandler : public drama::MessageHandler {
2.   public:
3.       MyObeyRescheduleHandler() {}
4.       ~MyObeyRescheduleHandler() {}
5.       drama::Request MessageReceived() override {
6.           MessageUser(
7.               "Hello World - from DRAMA 2, second entry");
8.           return  drama::RequestCode::End;
9.       }
10.  };
11.
12.  /*
13.   * Define a kick message handler.
14.   */
15.  class MyKickHandler : public drama::MessageHandler {
16.  public:
17.      MyKickHandler() {}
```

```
18.      ~MyKickHandler() {}
19.
20.      drama::Request MessageReceived() {
21.          MessageUser(
22.              "MyKickHandler: Kick received");
23.          return  drama::Request(
24.              drama::RequestCode::Wait, 1) ;
25.      }
26.
27.  };
28.
29.  // Action definition.
30.  class HelloAction : public drama::MessageHandler {
31.  public:
32.      HelloAction() {}
33.      ~HelloAction() {}
34.  private:
35.      MyKickHandler _myKick;
36.      MyObeyRescheduleHandler _reschedHand;
37.
38.      drama::Request MessageReceived() override {
39.          MessageUser(
40.              "Hello World - from DRAMA 2, first entry");
41.          PutObeyHandler(
42.              drama::MessageHandlerPtr(&_reschedHand,
43.                                       drama::nodel()));
44.          PutKickHandler(
45.              drama::MessageHandlerPtr(&_myKick,
46.                                       drama::nodel()));
47.
48.
49.          return  drama::Request(
50.              drama::RequestCode::Wait,20);
51.      }
52.  };
```

Below we see this code in action

```
>> ./exam5_4 &
>> ditscmd EXAMPLE5_4 HELLO&
DITSCMD_4a47:EXAMPLE5_4:Hello World - from DRAMA 2, first entry
>> ditscmd -k EXAMPLE5_4 HELLO
DITSCMD_4a48:EXAMPLE5_4:MyKickHandler: Kick received
>>
DITSCMD_4a47:EXAMPLE5_4:Hello World - from DRAMA 2, second entry
```

A request returned by kick handler will impact the rescheduling of the Obey itself. The kick handler may return `drama::RequestCode::End` to cause the action to end immediately. It can return `drama::RequestCode::KickReqNoChange` to not change the rescheduling of the Obey. This would typically be done if the Kick is rejected for some reason, in which case you may want to throw an exception or use the `drama::Request` constructor which allows you to return a DRAMA Status code to the parent task. In Example 5–4, the action is rescheduling again after a short delay.

5.6 Other ways of implementing handlers (e.g. as functions)

The above examples used objects that are subclasses of drama::MessageHandler to handle reschedule events. It is also possible to specify functions or methods to implement handlers.

There are variations of `drama::Task::Add()`, `drama::MessageHandler::PutObeyHandler()` and `drama::MessageHandler::PutKickHandler()` which take an argument of the type `drama::MessageReceiveFunction`. This is declared as

```
std::function<Request (MessageHandler *)>
```

By use of `std::function<>` and/or `std::bind()`, there are many ways to generate one of these. Example 5–5 shows two approaches. The first approach is actually the more complicated, specifying a method of an object as what we want invoked. At lines 26 to 29, we declare an objected named "f" which represents the `AltHandler()` method of the `HelloAction` class. Note – we don't care which class is involved here, it does not have to be the class `MessageReceived()` is part of.

Then at lines 33 and 34, we bind "f" and an object of the required class together and pass the result to `MessageHandler::PutObeyHandler()`. The placeholder is for the requirement argument. There will be a number of objects created behnd the sence here, but the result is that `AltHandler()` is invoked to handle the reschedule of the action.

WARNING: `AltHandler()` is part of `HelloAction` which is a sub-class of `drama::MessageHandler`. As a result, it has various methods available such as `MessageUser()` and `PutObeyHandler()`. You can't actually use those methods in `AltHandler()`, but must use the equivalent ones avialble in the "`MessageHandler *`" object passed as the argument. Using the wrong method here will cause a run-time exception, mentioning not being invoked as part of an action.

In that `AltHandler()`, we use the simpler alterative approach of specifying a function directly. Since the function in question – `HandlerFunc()`, has the required prototype, we can specify it directly. Otherwise we could use `std::bind()` to work around the prototype.

Example 5–5. Specifying functions/methods as handlers

```
1.  drama::Request HandlerFunc(drama::MessageHandler *mh)
2.  {
3.      mh->MessageUser("HandlerFunc invoked, action ending");
4.      return  drama::RequestCode::End;
5.  }
6.
7.  // Action definition.
8.  class HelloAction : public drama::MessageHandler {
9.  public:
10.     HelloAction() {}
11.     ~HelloAction() {}
12. private:
13.
14.     drama::Request AltHandler(drama::MessageHandler *mh) {
15.         mh->MessageUser(
16.            "HelloAction:AltHandler obey routine invoked");
17.         mh->PutObeyHandler(HandlerFunc);
18.         return  drama::RequestCode::Stage;
19.     }
20.
21.     drama::Request MessageReceived() override {
22.         MessageUser(
23.             "Hello World - from DRAMA 2, first entry");
24.         // Declare "f", a std::function of the right form.
```

```
25.          //   Needs the address of the method of interest.
26.          std::function<drama::Request (
27.              HelloAction*,
28.              drama::MessageHandler* )>
29.                  f(&HelloAction::AltHandler);
30.          // Put the handler, must bind f and this together.
31.          //   The std::placeholders::_1 allows for the argument
32.          //     to the call.
33.          PutObeyHandler(std::bind(
34.                              f, this, std::placeholders::_1));
35.
36.
37.          return  drama::RequestCode::Stage;
38.      }
39.  };
```

It may also be possible to specify a Lambda here. But please beware of the lifetime of any variables involved used.

5.7 Spawnable Actions

Normally, only one action of a given name can be running in a task at any one time. Most of the time, this makes sense. For example, an action to Initialise or shutdown (`EXIT`) a task, only one of each name should be running at a time. But there are some use-cases where it makes sense to have multiple actions of the same name running at the same time. In traditional C DRAMA, this has been rare but useful, as it requires the action to reschedule to make it work. With threads available in DRAMA2, this becomes more powerful. For example, you might have a `COMPUTE` action which does some calculations taking a long time based on arguments passed to the action. If the calculation is done in a thread on a multi-CPU machine, then you could run more then one of these at the same time.

In DRAMA, a "Spawnable" action is one that can have multiple independent invocations running simultaneously . They are more expensive to run then normal actions, but not dramatically so.

The `drama::Spawnable` interface class supports creating spawnable actions. You must sub-class this and implement the `Spawn()` and `ActionEnd()` methods. The implemented class is specified in a call to `drama::Task::AddSpawnable()` to add the action to the task.

The `drama::Spawnable::Spawn()` method is invoked each time the action is started. It must return a (shared) pointer object that is a subclass of `drama::MessageHandler`[5]. That object will be used to run the action.

When the action is complete, the `ActionEnd()` method is invoked so that any tidying up can be done. Note that this is often a null action, since if `Spawn()` returns a `std::shared_ptr()` to the object, it will be deleted automatically when the last reference is deleted.

Example 5–6 below shows how to do this. At line #31, the `HelloSpawnable` class is created as a subclass of `drama::Spawnable`. It's `Spawn()` method returns a

[5] For those who are are reading ahead – this can be a sub-class `of drama::thread::TAction`, that is, each spawned action can run in a different thread.

dynamically allocated object of class `SpawnStage1`, it is that implements the action. In this case, the action reschedules to `SpawnStage2` after 10 seconds.

Example 5–6 Spawnable Action Example

```
1.  /* Stage two of spawned action */
2.  class SpawnStage2 : public drama::MessageHandler {
3.  public:
4.      SpawnStage2() { }
5.      ~SpawnStage2() { }
6.  private:
7.      drama::Request MessageReceived() override {
8.          MessageUser("SpawnStage2 invoked");
9.          return  drama::RequestCode::End;
10.     }
11. };
12.
13. /* Stage one of spawned action */
14. class SpawnStage1 : public drama::MessageHandler {
15.     SpawnStage2 stage2;
16. public:
17.     SpawnStage1() { }
18.     ~SpawnStage1() { }
19. private:
20.     drama::Request MessageReceived() override {
21.         MessageUser("SpawnStage1 invoked");
22.         PutObeyHandler(
23.             drama::MessageHandlerPtr(
24.                 &stage2, drama::nodel()));
25.         return drama::Request(
26.             drama::RequestCode::Wait, 10) ;
27.     }
28. };
29.
30. /* Class which is used to spawn the action */
31. class HelloSpawnable : public drama::Spawnable {
32. public:
33.     HelloSpawnable() {}
34.     ~HelloSpawnable() {}
35. private:
36.     drama::MessageHandlerPtr Spawn() override {
37.         return drama::MessageHandlerPtr(new SpawnStage1());
38.     }
39.     void ActionEnd(drama::MessageHandlerPtr /*obj*/) override {
40.     }
41.
42. };
43.
44.
45. // Task Definition
46. class DramaExampleTask : public drama::Task {
47. private:
48.     HelloSpawnable HelloActionObj;
49. public:
50.     /*
51.      * Constructor.
52.      */
53.     DramaExampleTask(const std::string &taskName) :
54.         drama::Task(taskName) {
55.
56.         AddSpawnable("HELLO",
```

```
57.                         drama::SpawnablePtr(&HelloActionObj,
58.                                             drama::nodel()));
59.          // Standard simple EXIT action.
60.          Add("EXIT", drama::SimpleExitAction);
61.
62.      }
63.  };
```

In this example, since the action will run for 10 seconds, you can start a number of terminal windows and check that you can send it multiple times simultaneously

6 Implementing Actions using Threads.

The real power of DRAMA 2 comes from its ability to use threads. Much of the complexity of a traditional DRAMA task arises due to the need to keep the task responsive to messages, meaning it must avoid blocking. In traditional DRAMA tasks, I/O is implemented using say UNIX signals or the UNIX `select()` call, such that the action can return control to the main DRAMA loop whilst waiting for a response. Intensive computing must be broken up into components that can be interrupted and use of external non-DRAMA libraries for intensive computing or blocking I/O cause problems.

With threads, a DRAMA2 action implementation can avoid rescheduling but keep the task responsive to messages. The coding can often be far simpler, with a more obvious flow of control.

Most existing threaded code on Unix platforms is implemented using the POSIX `pthreads` library. This is a C language interface with some significant complexities. C++11 provides its own thread implementation and this is what is used by DRAMA2.

6.1 Major differences between pthreads and C++11

This list was taken from a paper presented to the 17th Geant4 Collaboration Meeting, by Marc Paterno (Fermilab). The paper is available from:

http://indico.cern.ch/event/199138/session/6/contribution/20/material/paper/0.pdf [6]

- `pthreads` is a C library, and was not designed with some issues critical to C++ in mind, most importantly object lifetimes and exceptions.
- `pthreads` provides the function `pthread_cancel` to cancel a thread. C++11 provides no equivalent to this.
- `pthreads` provides control over the size of the stack of created threads; C++11 does not address this issue.
- C++11 provides the class `std::thread` as an abstraction for a thread of execution.
- C++11 provides several classes and class templates for mutexes, condition variables, and locks, intending RAII[7] to be used for their management.
- C++11 provides a sophisticated set of function and class templates to create callable objects and anonymous functions (lambda expressions) that are integrated into the thread facilities.

The document mentioned above is a good quick introduction to C++11 threads.

It must be noted that the use of RAII to control thread resources is at the center of the design of the C++11 thread library and all of its facilities, and is also heavily used by DRAMA2.

6.2 Implementing C++11 Threads in any Application

There is nothing special about the function to be executed by a thread, except that it is good practice to prevent it from exiting on an exception, which would result in a call to

[6] This link may not open in Safari correctly. You might need to copy the text into Safari and open it that way.

[7] Resource Allocation Is Initialization

`std::terminate`. DRAMA Action implementations use `std::async()`, below, to run threads, which ensures any exceptions are handled correctly (action terminates and returns a bad DRAMA status, task does not terminate.).

The document linked to above provides a very quick intro to C++11 Threads, whilst the book "C++ Concurrency in Action", by Anthony Williams, ISBN 1933988770, provides a very detailed introduction. There are many in-between levels of information available on the web.

It should be noted that the basic DRAMA2 Threaded action implementation requires little knowledge from the user about C++111 threads.

6.2.1 Futures and async.

C++11 provides the class `std::future`, which can be used to return the value of a function run in its own thread of execution. Additionally, it can return any thrown exception (that is, the exception is transferred to the parent thread when it waits on the future). This is done by using template `std::async` is used to create the future. DRAMA uses this function to create action threads and it is suggested you consider it for any other threads you create.

6.2.2 Relationship to POSIX Threads.

On supported Unix style platforms (Linux, MacOsX), C++11 threads are usually[8] implemented using POSIX threads. The underlying POSIX thread associated with a C++11 thread is available using `std::thread::native_handle` member function. You may need to access this for cancellation, priority changes etc. The impact of cancellation on resources must be considered, since the C++11 code will presume destructors are run and if they are not, then locks may remain outstanding.

6.3 Basic DRAMA2 Approach

The DRAMA Internals and C interfaces were not written with threads in mind. A small number of global variables are used to maintain the task internal state; with the aim having been to ensure DRAMA C functions had simple interfaces. Update conflicts when accessing these variables is main area of concern.

The approach to working with threads used by DRAMA2 is based on one that proved successful in the DRAMA JAVA interface. One thread is made responsible for processing all DRAMA messages, in DRAMA2 this is the thread that invokes `drama::Task::RunDrama()`. Other threads can invoke DRAMA routines, but a lock is used to ensure no conflict. When `RunDrama()` is waiting for a message it does not need the lock, but takes it when processing a message.

But the DRAMA JAVA Interface did not allow actions to be implemented in threads. The threads were manly used to allow user interface threads to send DRAMA messages, that is, they were about access to DRAMA's UFACE context (User Interface context). A JAVA Thread could send messages, but all messages, including action implementations, were processed in thread that processed the DRAMA messages.

[8] How threads are implemented is determined by the C++ STL implementation you are using, but both GCC and Clang compilers, when running on Mac OS X and Linux, provide STL implementations that implement the using POSIX threads.

DRAMA2 allows actions to be run entirely within separate threads, and likewise, UFACE context message replies can be returned to the initiating thread. When a DRAMA2 Threaded action sends a message and wants to wait for the reply, it waits for a C++11 `std::condition_variable` to be notified, blocking the thread until the notification is received. The reply message will be received by `drama::Task::RunDrama()`, which will notify the condition variable, causing the thread to wake up.

The taking and releasing of the required lock, and the wait for the condition variable are all wrapped up such that user code simply sends a message, with the thread blocked until the reply is received or a timeout occurs.

The rest of this section deals with implementing actions using threads, but not with the sending of messages, that is covered in Section 7.

6.4 Implementing DRAMA Actions using threads

It is relatively simple to have your DRAMA action run in a separate thread. Instead of sub-classing `drama::MessageHandler`, your action implementation should be a sub-class of `drama::thread::TAction`. Instead of implementing `MessageHandler::MessageReceived`, it implements `TAction::ActionThread`. Example 6–1 below shows what this looks like:

Example 6–1. Implementing an action with a thread

```
1.  class HelloAction : public drama::thread::TAction {
2.  public:
3.      HelloAction(std::weak_ptr<drama::Task> theTask) :
4.          TAction(theTask) {}
5.      ~HelloAction() {}
6.  private:
7.
8.      void ActionThread(const drama::sds::Id &) override  {
9.          MessageUser("Hello World - from threads DRAMA 2");
10.     }
11. };
12.
13. // Task Definition
14. class DramaExampleTask : public drama::Task {
15. private:
16.     HelloAction HelloActionObj;
17. public:
18.     /*
19.      * Constructor.
20.      */
21.     DramaExampleTask(const std::string &taskName) :
22.         drama::Task(taskName), HelloActionObj(TaskPtr()) {
23.
24.         Add("HELLO", drama::MessageHandlerPtr(
25.                 &HelloActionObj, drama::nodel()));
26.         // Standard simple EXIT action.
27.         Add("EXIT", &drama::SimpleExitAction);
28. };
```

It should be noted that the `thread::TAction` constructor needs the drama::Task object address passed to it, see line #3 and line #21. Otherwise this is easy, and very similar to Example 1–1.

The `thread::TAction` class provides its own `MessageUser` method, which sets up the DRAMA context correctly such as to emulate `MessageHandler::MessageUser` (that is, a parent action sees the message as having come from the action). No request needs to be returned as the action completes when the thread completes.

In this example, `TAction::ActionThread` is implemented within its own thread. The DRAMA task will remain responsive whilst that thread is running. Of course, in this example, it completes quickly anyway, but it is easy to demonstrate it remains responsive: For example, add some code to cause the thread to block (say using `std::this_thread::sleep_for`) and then send a second action (any name, anything but EXIT will be rejected immediately) to it. If you try this with Example 1–1, the response won't occur until the action wakes up and completes.

Note that if you do send EXIT to the task whilst the action thread is running, the task will not EXIT until the thread completes. This is because the action thread must be "joined" to the main thread before the task can exit[9].

6.5 Accessing Action Arguments

Any argument to action the is accessible from the drama::sds::Id argument supplied to `drama::TAction::ActionThread`. Example 6–2 is a minor modification of Example 6–1 showing this.

Example 6–2. Threaded action argument

```
1.      void ActionThread(const drama::sds::Id &id) override  {
2.          if (id)
3.          {
4.              MessageUser(
5.                  "Action was supplied with an argument");
6.              id.List();
7.          }
8.          else
9.          {
10.             MessageUser(
11.                 "Action was NOT supplied with an argument");
12.         }
13.     }
```

It must be noted that in order to ensure any SDS argument is valid when the thread is running, `TAction::ActionThread` is supplied with a copy of the item supplied to the action, since the item supplied to the action no longer available when DRAMA finishes processing the message. This can impact program performance if the SDS argument structure is very large, and it is possible other designs might be appropriate when an action expects very large arguments.

6.6 Kick Messages.

It is possible to kick an action implemented with threads, but the action must be waiting for the kick, using one of the `drama::thread::TAction::WaitForKick*` methods. `WaitForKick` will wait without a timeout, or you can request a timeout in a certain period with `WaitForKickTimeoutIn` or at a certain time with `WaitForKickTimeoutAt`.

[9] You could detach your thread, which would avoid this, but then DRAMA will have problems when the action completes before the task, as it expects to join the thread to recover status information.

All of these methods allow you to access any SDS argument to the Kick message. As per the original obey, the SDS argument is actually a copy of the argument in the message, so if the size is likely to be large, consider carefully if threaded actions are the appropriate approach. To fetch the argument, you need to supply a `drama::sds::IdPtr` item which is a `std::shared_ptr` for an `sds::Id`. This is needed to ensure the item is released correctly only when everything is finished accessing it.

Example 6–3 is a variation on Example 6–1 and implements a 15 second wait for a kick to occur. If kicked, it checks if there is an argument and if there is, outputs it. `WaitForKickTimeoutIn` returns true if the kick was received, false if it timed out. This version of `WaitForKickTimeoutIn` takes an unsigned integer number of seconds, but a version which accepts a `std::chrono::duration` is also available.

Example 6–3. Kicking a threaded action

```
1.   void ActionThread(const drama::sds::Id &) override  {
2.
3.           MessageUser("Waiting for kick.");
4.           // Item used to receive Kick argument.
5.           drama::sds::IdPtr
6.               Arg{std::make_shared<drama::sds::Id>()};
7.           // Wait for a kick, timeout of 15 seconds.
8.           if (WaitForKickTimeoutIn(15, &Arg))
9.           {
10.              // Kick received.
11.              if (*Arg)
12.              {
13.                  std::string MyArgument;
14.                  Arg->Get("Argument1", &MyArgument);
15.                  MessageUser("Received Kick with argument \"" +
16.                              MyArgument +
17.                              "\"");
18.              }
19.              else
20.              {
21.                  MessageUser("Received Kick without argument");
22.              }
23.          }
24.          else
25.          {
26.              MessageUser("Wait for kick timed out");
27.          }
28.      }
```

6.7 User initiated threads

Threaded actions become far more interesting when they are themselves running multiple threads. How the action thread and any child threads interact is up to the programmer, but it is possible for child threads of an action thread to use DRAMA facilities in the context of the action passing the action object to the child thread. Example 6–4 shows a thread starting a child thread. The "`HelloAction`" pointer is passed to the child thread so it can access DRAMA features. Both threads will output DRAMA messages and waiting for a kick message. The one kick message will wake up all waiting threads.

Example 6–4. Theaded action with child thread

```
1.   class HelloAction : public drama::thread::TAction {
2.   public:
```

```
3.      HelloAction(drama::Task *theTask) : TAction(theTask) {}
4.      ~HelloAction() {}
5.  private:
6.
7.      void ActionThread(const drama::sds::Id &) override  {
8.
9.          MessageUser("Starting second thread");
10.         std::thread childThread(SecondThread, this);
11.         MessageUser("First thread waiting for kick.");
12.         // Item used to receive Kick argument.
13.         drama::sds::IdPtr
14.             Arg(std::make_shared<drama::sds::Id()};
15.         // Wait for a kick, timeout of 15 seconds.
16.         if (WaitForKickTimeoutIn(15, &Arg))
17.         {
18.             // Kick received.
19.             if (*Arg)
20.             {
21.                 std::string MyArgument;
22.                 Arg->Get("Argument1", &MyArgument);
23.                 MessageUser(
24.                     "First thread kicked with argument \"" +
25.                     MyArgument +
26.                     "\"");
27.             }
28.             else
29.             {
30.                 MessageUser(
31.                     "First thread kicked without argument");
32.             }
33.         }
34.         else
35.         {
36.             MessageUser("Wait for kick timed out");
37.         }
38.         MessageUser("First thread waiting to join child.");
39.         childThread.join();
40.         MessageUser("Threads have joined, action complete.");
41.     }
42. };
43. static void SecondThread(HelloAction *action)
44. {
45.     action->MessageUser(
46.         "Second thread running, waiting for kick");
47.     drama::sds::IdPtr Arg{std::make_shared<drama::sds::Id>()};
48.     if (action->WaitForKickTimeoutIn(10, &Arg))
49.     {
50.         action->MessageUser("Second thread, received kick.");
51.         if (*Arg)
52.         {
53.             std::string MyArgument;
54.             Arg->Get("Argument1", &MyArgument);
55.             action->MessageUser(
56.                 "Second thread kicked with argument \"" +
57.                 MyArgument +
58.                 "\"");
59.         }
60.         else
61.         {
62.             action->MessageUser(
63.                 "Second thread kicked without argument");
```

```
64.             }
65.         }
66.         else
67.         {
68.             action->MessageUser(
69.                 "Second thread, wait for kick time out.");
70.         }
71.     }
```

What Example 6–4 fails to do is to correctly handle any exceptions thrown in the child thread. As written, if the child throws an exception, the program is terminated. A better solution might be to run the thread via `std::async`. If run this way, exceptions can be transferred to the parent thread. This experiment is left to the reader and I don't want to have all the fun[10].

6.7.1 POSIX Threads

In Example 6–4, the child thread is a C++11 thread. There is nothing to stop you using a POSIX thread (or any other supported thread type). The only real condition is that the C++11 Mutex works, which is not thread architecture specific.

As C++11 threads are normally implemented on Unix platforms with POSIX threads, you can use `std::thread::native_handle` to access the underlying POSIX thread. This might allow you to adjust the priority of the thread or to cancel it. You need to think very carefully about cancelling threads if you think you need to. It is likely a POSIX Asynchronous Thread Cancelation will leave resources in undefined states, and even Synchronous Thread Cancelation can cause problems, as it is undefined if C++ destructors are invoked[11]

Be warned that if you change the priority of threads, you must give careful consideration to the potential consequences to avoid "priority inversion" issues[12].

6.8 Kicking threads that are blocked.

One of the benefits of implementing actions using threads is the ability to invoke code that will block without making the task unresponsive. There are many possible causes, a blocking `read()`, a CPU intensive loop, a wait on a mutex or semaphore. For any of these events, it is still desirable in many cases to be able to unblock the thread and cause the action to complete.

Such a thread could be cancelled using `pthread_cancel()`, but this presents a host of potential issues, including not tidying up the DRAMA action correctly. It was realized[13] that

---

[10] I suggest modifying the example to throw an exception from the child, demonstrating the failure. Then rewrite appropriately to see the exception transferred through to the parent thread and reported via DRAMA.

[11] Some information suggests that some systems may do this.

[12] In POSIX threads, the `pthread_mutexattr_setprotocol()` function can be used to enable priority inheritance. It is unclear if this is supported in Mac OS X and linux, if it is the default and what C++11 does with its own mutexs (probably based on POSIX mutex).

[13] K. Shortridge and T. J. Farrell, "Progress in cancellable multi-threaded control software", Proc. SPIE 7740, Software and Cyberinfrastructure for Astronomy, 774029 (July 19, 2010); doi:10.1117/12.856217; http://dx.doi.org/10.1117/12.856217

whilst there is no reliable way of unblocking a thread in general, there is often some way of unblocking a particular operation. For example, a blocking `read()` can be aborted by closing the file descriptor. For a CPU loop, it may be practical to check a cancel flag regularly. For a semaphore wait, sending a unix signal to the thread will abort the wait. As a result, what is needed is a well-defined way for a Kick message to an action to invoke user-defined code when the action is blocked.

The DRAMA2 solution is the `drama::thread::KickNotifer` class. From an action thread, create a `KickNotifier` object or a sub-class just before code that will block. This results in a new thread being created. This new thread will process any kick messages received by the action. User code can ask if a kick has been received using `WasKicked()` or a sub-class can provide the `Kicked()` method, which is invoked when a kick is received. The `KickNotifier` destructor will destroy the thread created by the constructor.

A user implementation of the `KickNotifier::Kicked()` method may do what is necessary to unblock the action thread.

Example 6–5 shows the use of `KickNotifier` in the simpler case, where a hard CPU loop that can check for a flag occasionally is invoked. It is left to the reader to try the more complex case of sub-classing `KickNotifier`, but authors should beware that `KickNotifier::Kicked()` is invoked in the thread created by `KickNotifier`, which may introduce data access problems requiring locks (see 6.10 for help with this).

Example 6–5. Kicking a blocked thread.

```
1.   class HelloAction : public drama::thread::TAction {
2.   public:
3.       HelloAction(std::weak_ptr<drama::Task> theTask) :
4.           TAction(theTask) {}
5.       ~HelloAction() {}
6.   private:
7.
8.       void ActionThread(const drama::sds::Id &) override  {
9.           MessageUser("Action Starting");
10.          drama::thread::KickNotifier unblockObj(this);
11.          for (unsigned i = 0; i < 50 ; ++i)
12.          {
13.              for (unsigned j = 0; j < 100000000 ; ++j)
14.              {
15.
16.              }
17.              MessageUser("Alive");
18.              if (unblockObj.WasKicked())
19.              {
20.                  MessageUser("Action Was kicked.");
21.                  return;
22.              }
23.          }
24.          MessageUser("Action complete");
25.      }
26.  };
```

`KickNotifier` does provide the ability the process multiple kick messages, as well as to read the argument to the kick message, both via the `Kicked()` method implementation. The main outstanding case is the hard CPU loop which cannot be modified to check for a flag.

6.9 Trigger Messages, Output Arguments

The `drama::thread::TAction` also has a SendTrigger() method, which works in the same way as `drama::MessageHandler`'s method of the same name (see section 4.5.1).

Simliarly, there are also `drama::thread::Taction::SetReturnArg` methods, working in the same way as `drama::MessageHandler`'s method of the same name (see section 1). But it should be noted that this can only be sent when the action thread has compeleted.

6.10 Other ways of specifying handlers (e.g. as functions)

In section 5.6 we saw how non-threaded action handlers could be implemented as functions rather then using objects. The same applies to action threads. In this case, there is a method `drama::Task::AddTA()` which takes an argument of type `thread::ThreadActionFunction`. This is declared as

```
std::function<void (drama::thread::TAction *, const drama::sds::Id &)> ;
```

Example 6–6 below shows one approach to doing this. At line #21, we add the action, specifying a function of the correct prototype to use. The implementation function is actually very simple code from lines 1 to 8. The important feature is that the first argument must be used to access DRAMA.

Example 6–6. Implementing a thread action with a function.

```
1.   static void HelloThreadFunc(
2.       drama::thread::TAction *threadAct,
3.       const drama::sds::Id &obeyArg)
4.   {
5.
6.       threadAct->MessageUser(
7.           "Action thread function running");
8.   }
9.
10.
11.  // Task Definition
12.  class DramaExampleTask : public drama::Task {
13.  public:
14.      /*
15.       * Constructor.
16.       */
17.      DramaExampleTask(const std::string &taskName) :
18.          drama::Task(taskName) {
19.
20.          // HELLO action, implemented via thread.
21.          AddTA("HELLO", HelloThreadFunc);
22.          // Standard simple EXIT action.
23.          Add("EXIT", drama::SimpleExitAction);
24.
25.      }
26.  };
```

As per section 5.6, there are many ways of working with this. For example, `std::bind()` could used to bind arguments to functions with more arguments then standard.

There is also an overload of `drama::Task::Add()` which takes a function of type `MessageReceiveFunction`, in this case, it is a non-threaded action we are implementing via an action. `MessageReceiveFunction` is declared as

```
std::function<drama::Request (drama::MessageHandler *)> ;
```

6.11 Locks - Working with older DRAMA interfaces or other shared data

Threaded code must take a lot of care if it wishes to work with older DRAMA C or C++ Interfaces. User code must ensure that the lock to access DRAMA internal is taken and that the DRAMA "internal context" is set correctly. The later means that the DRAMA system knows which action is currently being executed.

In almost all cases, this can be done by constructing an object of type `drama::thread::AccessDrama` before executing the traditional DRAMA code you wish to execute. This uses RAII to ensure the DRAMA internals are restored correctly. Pass the action's `drama::TAction` object address to the `AccessDrama` constructor. From the point one of these is constructed, until the destructor is run, the DRAMA lock is taken and the DRAMA Context is set to that of the action. Since the lock is taken, you can also use this object to lock access to any shared data of interest.

Alternatively, you may want to take the DRAMA lock but not switch the context. This might be usefull for using the DRAMA lock around resource contention issues in your own program. The DRAMA lock itself is available via `drama::Task::Lock()`. You should create an object of type `drama::Task::guardType` (currently a `std::lock_guide` type) or `drama::Task::uniqueLockType` (currently a `std::unique_lock` type) as required, with this lock as its argument, using RAII, to protect your code.

It should be noted that the DRAMA lock is a recursive lock, based on `std::recursive_timed_mutex`, so you have no problem taking it if the thread already has it.

Note that it is very important that:

1. You hold the DRAMA lock for as short a period as possible to ensure other threads are not blocked for longer then necessary.
2. That careful consideration is given to the potential consequences if thread priorities are changed from normal, so as to avoid "priority inversion" issues[12].
3. That careful consideration is given to the potential consequences if your application implements its own lock rather then using the DRAMA lock.

In cases 2 and 3 above, the risk is deadlock!

6.12 Interactions with signals

Threads and Unix Signals do not work well together. A traditional UNIX signal will be delivered to one thread only, but may be directed towards any thread that has not blocked the signal. If starting an application from scratch, you would avoid using UNIX signals in a program with threads. But DRAMA has been around for a long time and does use UNIX signals, in particular, SIGALRM.

The normal way to deal with Threaded applications that might receive signals is to block signals in all threads but the one intended to receive them. In the DRAMA2 case, the thread invoking `drama::Task::RunDrama` should receive the signals. Threads started by DRAMA will have all signals blocked, and a child thread inherits its signal mask from its parent, so Example 6–4 is safe without further work.

If creating threads yourself, from outside a DRAMA Action thread, you should ensure all signals in the thread are blocked. First, before creating the thread, construct and object of type `drama::thread::SignalBlocker`. This class's constructor will block signals and its destructor will restore the signal state. This is done to ensure you don't get a signal whilst the thread is being created, leaving it in an undefined state. Then, as the first line in you thread, invoke `drama::thread::BlockSignals()`, which blocks all signals to the thread.

6.13 Thread Programming Issues

In addition to the correct usage of locks (See section 6.11, above) there are other issues you must beware of with thread program. This section lists some which have impacted DRAMA programs.

6.13.1 Threaded Programs not exiting when expected.

This can happen if you if have non-detected threads still running They will block program exit in various ways.

6.13.2 Delayed Exception delivery

Consider the following code segment (which won't compile, but is close)

```
1.   void test(drama::thread::TMessHandler *tMessHandler,
2.              drama::Path *task)
3.   {
4.       // Start a thread which sends an OBEY PROCESS
5.       std::future<void> processFuture = std::async(
6.           std::launch::async, [this, task] {
7.               RunProcess(tMessHandler task); } );
8.
9.       // Send another action, whilst the first is still running
10.      task->Obey(tMessHandler, "SET_WINDOW");
11.      ...
12.      task->Kick(tMessHandler, "PROCESS");
13.      // Wait for PROCESS action to complete.
14.      processFuture.get();
15.
16.  }
```

At line #4, this code is starting a thread, which happens to obey an action named `PROCESS`. That action will run until kicked.

At line #10, the action `SET_WINDOW` is sent to the task. Some other things are done and then at line #12, the `PROCESS` action is kicked. This will cause the `PROCESS` action to complete, hence the `RunProcess()` thread completes and `processFuture.get()` will return then return.

But, if for example, the `Obey` of `SET_WINDOW` throws an exception, this code won't complete. The throw of the exception triggers the destructor of processFuture(). That destructor won't complete until the thread has completed. Since in this case, the `PROCESS` action is not kicked, the thread does not complete. The program hangs whilst running the destructors of the code block.

So how might one handle this suitation: The code below shows one possibility. The Obey `SET_WINDOW` and operations before the Kick are wrapped in a try block. The exception

handler now does the kick and the get of the future (as does the non-exception code – C++ really needs a "`finally`" clause to a try block.)

```
1.   void test(drama::thread::TMessHandler *tMessHandler,
2.              drama::Path *task)
3.   {
4.       // Start a thread which sends an OBEY PROCESS
5.       std::future<void> processFuture = std::async(
6.           std::launch::async, [this, task] {
7.               RunProcess(tMessHandler task); } );
8.
9.       try
10.      {
11.          // Send another action, whilst the first is still running
12.          task->Obey(tMessHandler, "SET_WINDOW");
13.          ...
14.      }
15.      catch (...)
16.      {
17.          task->Kick(tMessHandler, "PROCESS");
18.          processFuture.get();
19.          throw;
20.      }
21.      task->Kick(tMessHandler, "PROCESS");
22.      // Wait for PROCESS action to complete.
23.      processFuture.get();
24.
25.  }
```

What the above doesn't handle is an exception thrown by the Kick. That is hard to deal with – particularly as it probably means the thread won't complete. A timeout on the underlying `OBEY PROCESS` combined with ignoring any exception from Kick may provide an escape. As you can see, threaded code can get complicated!

6.13.2.1 Delayed until an destructor is run

A potentially hard to deal with example of this is where the future::get() operation or similar is being executed within a destructor. In an example like the above, unless it is caught, the program will be terminated. See section 3.4 for more details.

7 Sending Messages to tasks

In all the examples in the previous sections of this document, we have been dealing with tasks that receive messages and send replies to those messages. These are in effect tasks providing services within a DRAMA system. They might control hardware or implement CPU intensive processing. In this section, we deal with the message originators – tasks that initiate messages to other tasks and process the replies.

Such tasks are used to implement user interfaces and to implement system "control tasks", used to coordinate a set of tasks. The 2dF system is the largest existing example of such a set of tasks. The 2dF control task runs a set of about 30 tasks, some of which are also control tasks running smaller sets of tasks.

This section will deal only with the "control task" case, with user interfaces covered in section 8.

7.1 DRAMA Task loading and Networking configuration

DRAMA implements a tasking distributed system, with DRAMA tasks able to be run on various machines. Examples so far have dealt only with a pair of tasks ("DRAMAHELLO" examples, and `ditscmd`) running on the same machine. Examples in this section will take advantage of the DRAMA task loading facilities to load tasks, if needed. You could also modify these examples to distribute the tasks across different machines.

To enable both loading tasks and distribution of DRAMA systems across various machines, you must run the DRAMA "networking" tasks. This is done by executing the "`dits_netstart`" command on each machine involved, under the same user name[14]. They can be shutdown with "`dits_netclose`".

7.2 DRAMA 2 Message sending basics

All message sending in DRAMA 2 tasks is done from action threads[15] (see section 6).

All message sending routines *will block the thread* until the message is complete, so if you wish to send multiple messages from the one action simultaneously, you will need to create a separate child thread for each simultaneous message.

The `drama::Path` class is the key class involved. In C language DRAMA, a "path" is a channel for communication with another task. In DRAMA2, the `drama::Path` class provides facilities for sending messages and receiving replies along a path.

In the examples below, we will be using an example target (Server) program implemented in example code file `example_server.cpp`. The actual source of this is not particularly relevant, so will not be examined, but you are welcome to look. This program, by default, uses the task name "SERVER_TASK". Setting the first command line argument, which is done in some examples, can change the task name.

7.3 Path constructor and related methods.

[14] DRAMA Provides alternatives to using the same user name on the different machine, the direct specification of the communications port number via an IMP Startup file. See the DRAMA web pages for details.

[15] Actually this is not really true – user interface code may not use action threads, but we leave that until section 8. But the rest of what is described here works in that case as well.

The most commonly used `drama::Path` constructor takes four arguments, two of which have defaults. Details are in the table below.

Argument	Type	Description
`theTask`	`drama::Task *`	The DRAMA2 task object.
`name`	`std::string`	The name of the task you with to talk to, presuming it is already running. If we load the task then the name it registers as will be used instead.
`host`	`std::string`	The name of the host on which to load the task or on which it is running. This is only used if the task is not already running on the local machine, or is otherwise known[16] to the local machine. Defaults to an empty string. Alternatively, you can include the node name with the target task name using the `name@host` format.
`file`	`std::string`	The file from which to load the task, if it is not already running. It will be loaded on the host specified above.

Once you have constructed such an object, you are able to "Get" the path. Getting a path involves loading a task if necessary, and then opening the communications channel to the task. One you have gotten the path to a DRAMA task, communications becomes very efficient.

Various other methods can be used to set values applied when the program is loaded or when the path is set up, see sections 7.4.1 and 7.4.2 for details.

There are two other constructors available. One takes only a "drama::Task *" argument; it creates a path to the task itself, allowing you to send messages to your task using DRAMA, which is sometimes useful.

The remaining constructor takes a traditional DRAMA `DitsPathType` variable as the second argument, allowing `drama::Path` to take over an existing path.

It should be noted that only in the first case, is it necessary to "Get" the path.

7.4 Loading tasks and Getting Paths

In most cases, the first thing you need to do with a path is to "Get" it. This runs the processes involved in opening communications with the target task, but may also include loading the task. Having constructed a drama::Path object, you need only execute the "GetPath" method, specifying as an argument, the drama::TAction object of your threaded action

[16] A remote task is known locally if another task is already communicating with it.

implementation. This method will block until the task is loaded (if needed) and the path is opened for communications.

Example 7–1 below shows how this is done. The name of the task we are getting the path to is "SERVER_TASK" and if no task of that name is specified, then the program "./example_server" is loaded and presumed to be the task in question. The method server.GetPath() will block until the operation is complete, so when it returns, you can send messages.

Example 7–1. Loading tasks and getting the path

```
1.   class RunAction : public drama::thread::TAction {
2.
3.   public:
4.       RunAction(std::weak_ptr<drama::Task> theTask) :
5.           TAction(theTask), _theTask(theTask) {}
6.       ~RunAction() {}
7.   private:
8.       std::weak_ptr<drama::Task>   _theTask;
9.       void  ActionThread(const drama::sds::Id & /*obeyArg */) {
10.          drama::Path server(_theTask,
11.                             "SERVER_TASK", "",
12.                             "./example_server");
13.          server.GetPath(this);
14.          MessageUser("Have gotten path");
15.      }
16.  };
17.
18.  class ClientTask : public drama::Task {
19.
20.  private:
21.      RunAction RunActionObj;
22.  public:
23.      /**
24.       * Constructor, from here we add actions
25.       */
26.      ClientTask(const std::string &taskName) :
27.          drama::Task(taskName), RunActionObj(TaskPtr()) {
28.
29.          Add("RUN", drama::MessageHandlerPtr(
30.                  &RunActionObj, drama::nodel()));
31.          Add("EXIT", drama::SimpleExitAction);
32.      }
33.      ~ClientTask() {
34.      }
35.  };
```

To run Example 7–1, first execute "dits_netstart". Then run the example client with "./exam7_1 &". Finally, use "ditscmd" to send RUN actions to it. E.g.:

```
> dits_netstart
> ./exam7_1 &
> ditscmd CLIENT RUN
DITSCMD_3296:CLIENT:Loading task SERVER_TASK from "./example_server"
DITSCMD_3296:CLIENT:Task loaded
DITSCMD_3296:CLIENT:Have gotten path
> ditscmd CLIENT RUN
DITSCMD_379a:CLIENT:Have gotten path
>
```

Note that if "SERVER_TASK" is already running, then the load operation is not needed. The `drama::Path::LogLoad()` method can be used to disable the message output when loading a task, if required.

The following operation order should be noticed.

```
Get Path.
     If Get Path fails and have a file name.
          Load Task
          If Load Path Ok
               Get Path
```

That is, an attempt is made to get the path. If that fails, and attempt is made to load the task and if the succeeds, a second attempt is made to get the path. If at any stage

7.4.1 drama::Path methods that impact task loading.

The table below shows the methods, which if executed before `drama::Path::GetPath`, will impact task loading, if task loading is done.

Name	Description
`drama::Path::SetHost`	Set the name of the host on which to load the task.
`drama::Path::SetArgument`	Set the command line argument string to the load.
`drama::Path::SetFile`	Set the file to load the program from. You can use the `<envvar>:<filename>` format if required.
`drama::Path::SetPriority`	Set the program priority.
`drama::Path::SetNames`	Used to control the inheritance of environment variables.
`drama::Path::SetSymbols`	Specific to target programs running on the VMS operating system. See the documentation for details.
`drama::Path::SetProg`	Specific to target programs running on the VMS operating system. See the documentation for details.
`drama::Path::LogLoad`	Enable/disable logging of task loading via MsgOut() (MessageUser()) messages.

7.4.2 drama::Path methods that impact Get Path operations.

The table below shows the methods, which if executed before `drama::Path::GetPath`, will the getting of the path.

Name	Description
`drama::Path::SetName`	Set the name of the task. If the program

	is loaded and registers using a different name then that name will be used.
`drama::Path::SetHost`	Set the name of the host on which to find the task, if it is not already known to the local machine.
`drama::Path::SetFlowControl`	Use to enable flow control on the path. If invoked, communications are run differently when buffers fill up. See DRAMA documentation for more detail.
`drama::Path::SetBuffers`	Set the DRAMA Path buffers to be used. This is used to specify the size of the communications buffers, which must be large enough for the biggest set of unprocessed messages.

7.5 Loading non-DRAMA programs.

The "`drama::Path`" class could be used to load a non-DRAMA program. The `GetPath()` method will block until the program exits. If it exits without error, then a `drama::Exception` will be thrown indicting, in effect, that it didn't register with DRAMA. This will have the status code DRAMA2__PATH_EXITONLOAD. Otherwise it will throw an exception due to the failure of the program.

Alteratives to drama::Path are the `drama::thread::RunProgram()` and `drama::thread::RunProgramWaitUntil()` functions. These do the same thing but don't need a `drama::Path()` object and don't throw an exception on the normal exit of the program. They are the preferred approach to loading non-DRAMA programs.

It must be noted that there is no way for your task to know a non-DRAMA program has started successfully (other then something you might implement yourself). DRAMA knows if the program did not execute (say the file does not exist) and it knows it has exited, but can't work out if it is running, since the operating system doesn't provide the information. For DRAMA programs, a message is arranged when the program initializes DRAMA which allows the loader to know the program is now running.

7.6 Sending Obey Messages

To send a command to another task, you send an "Obey" message, specifying the thread action object address and name of the action to invoke. Optional parameters to the method include any argument to the action, a spot for the return of any argument in the completion message and an event processor object, which we examine in section 7.8. Example 7–2 expands the "`ActionThread`" method from Example 7–1 to send an Obey message, of the Action named "ACTION1".

Example 7–2. Sending an Obey message

```
1.      void  ActionThread(const drama::sds::Id & /*obeyArg */) {
2.           drama::Path server(_theTask,
3.                              "SERVER_TASK", "",
4.                              "./example_server");
5.           server.GetPath(this);
6.           MessageUser("Have gotten path, will send obey");
7.
```

```
8.          server.Obey(this, "ACTION1");
9.
10.         MessageUser("Subsidiary Action completed");
11.
12.     }
```

7.6.1 Adding an argument to the Obey message

The third parameter to the `drama::Path::Obey` method is the SDS ID of an argument to the message. Anything that can be constructed in SDS can be sent, as long as the message buffer sizes are large enough. (The buffer sizes can be set using the `drama::Path::SetBuffers` method, before the `GetPath` method is invoked, the defaults detail with most small message cases). Example 7–3 is a modification of Example 7–2, which sends an argument.

Example 7–3. Sending an Obey with an argument.

```
1.   void  ActionThread(const drama::sds::Id & /*obeyArg */) {
2.          drama::Path server(_theTask,
3.                             "SERVER_TASK", "",
4.                             "./example_server");
5.          server.GetPath(this);
6.          MessageUser("Have gotten path");
7.          drama::sds::Id messageArg(
8.              drama::sds::Id::CreateArgStruct());
9.          messageArg.Put("Argument1", "Hello from example 7.3");
10.         server.Obey(this, "ACTION1", messageArg);
11.         MessageUser("Subsidiary Action completed");
12.     }
```

If you need to specify the fourth or fifth parameter to `drama::Path::Obey`, but don't want to specify an argument to the Obey message, you need to specify a null SDS item as the third argument, using `drama::sds::Id::CreateNullItem`.

7.6.2 Recovering the completion message argument value from an Obey.

The fourth parameter to the `drama::Path::Obey` method is used to retrieve any argument to the action completion message. If interested in the argument, you must pass the address of a `drama::sds::IdPtr` item, which is a `typedef` of `std::shared_ptr<drama::sds::Id>`, that is; a shared pointer to a `sds::Id`. A shared pointer is used to avoid copying the SDS item (a second time).

Example 7–4 below is a rework of Example 7–2 to show how to access the completion message argument. The item "`returnedArg`" is declared at line #7 and passed by pointer at line #10. You should check a value was actually returned before using it, as per line #11, and then access the SDS item by dereferencing the pointer as shown at line #14

Example 7–4. Accessing Obey message completion argument

```
1.     void  ActionThread(const drama::sds::Id & /*obeyArg */) {
2.          drama::Path server(_theTask,
3.                             "SERVER_TASK", "",
4.                             "./example_server");
5.          server.GetPath(this);
6.          MessageUser("Have gotten path");
7.          drama::sds::IdPtr returnedArg;
8.          server.Obey(this, "ACTION1",
```

```
9.                      drama::sds::Id::CreateNullItem(),
10.                     &returnedArg);
11.         if (*returnedArg)
12.         {
13.             std::string returnedAsString;
14.             returnedArg->Get("Argument1", &returnedAsString);
15.             MessageUser(
16.                 "Subsidiary Action completed, returned arg = \""
17.                 + returnedAsString + "\"");
18.         }
19.         else
20.             MessageUser(
21.                 "Subsidiary Action completed, no arg returned");
22.     }
```

If you need to specify the fifth parameter to `drama::Path::Obey`, but don't want to retrieve any completion argument, you need to specify a `nullptr` as the fourth argument,

7.7 Sending Kick Messages

The `drama::Path::Kick` method allows you to send Kick messages. The calling sequence and the way the method is used are exactly like `Path::Obey`.

Whilst sending kick messages is easy, our example must be changed a fair bit. Rather then sending `ACTION1` to the `SERVER_TASK`, it sends `ACTION2`, which will wait 10 seconds before completing, and it accepts Kick messages. If `ACTION2` is sent a Kick message with an argument (any value) it causes the action to complete immediately. If no argument is supplied, the action will wait an extra 10 seconds before completing.

The example code implements a second action, named "`KICKIT`", which you can use to send the Kick message. In order to ensure the "`KICKIT`" action does not need to get the path to "`SERVER_TASK`" a second time, the relevant `drama::Path` variable is now kept within the `ClientTask` object, rather then in the `ActionThread` methods. This was the cause of much of the structure change, since the action classes must reference the `ClientTask` class, which must reference the action classes.

Example 7–5, below, shows all the relevant bits of the example. The main point of interest, the sending of the Kick message, is at line #84. Note how the input argument to the action is passed directly to the `drama::Path::Kick` method. If no argument was supplied, then nothing will be sent. If it is supplied, it is sent directly on with the Kick message.

Example 7–5. Sending kick messages

```
1.  /*
2.   * Note - Definitions of constructors and ActionThread
3.   * methods must be after ClientTask is defined.
4.   */
5.  class RunAction : public drama::thread::TAction {
6.
7.  public:
8.      RunAction(std::weak_ptr<drama::Task> theTask);
9.      ~RunAction() {}
10. private:
11.     std::weak_ptr<drama::Task> _theTask;
12.     void  ActionThread(const drama::sds::Id & /*obeyArg */);
13. };
14.
15. class KickItAction : public drama::thread::TAction {
```

```
16.
17.  public:
18.      KickItAction(std::weak_ptr<drama::Task> theTask);
19.      ~KickItAction() {}
20.  private:
21.      std::weak_ptr<drama::Task> _theTask;
22.      void  ActionThread(const drama::sds::Id & /*obeyArg */);
23.
24.      std::shared_ptr<ClientTask> GetTask() {
25.          return std::shared_ptr<drama::Task>(_theTask)->
26.              TaskPtrAs<ClientTask>();
27.      }
28.  };
29.
30.  class ClientTask : public drama::Task {
31.
32.  private:
33.      RunAction _runActionObj;
34.      KickItAction _kickItActionObj;
35.      drama::Path _serverPath;
36.  public:
37.      drama::Path &ServerPath() {
38.          return _serverPath;
39.      }
40.      /* Constructor */
41.      ClientTask(const std::string &taskName) :
42.          drama::Task(taskName),
43.          _runActionObj(TaskPtr()),
44.          _kickItActionObj(TaskPtr()),
45.          _serverPath(TaskPtr(), "SERVER_TASK",
46.                      "", "./example_server") {
47.
48.          Add("RUN", drama::MessageHandlerPtr(
49.                  &_runActionObj, drama::nodel()));
50.          Add("KICKIT", drama::MessageHandlerPtr(
51.                  &_kickItActionObj, drama::nodel()));
52.          Add("EXIT", drama::SimpleExitAction);
53.      }
54.      ~ClientTask() {
55.      }
56.  };
57.
58.  /* Define RunAction and KickItAction constructors */
59.  RunAction::RunAction(
60.      std::weak_ptr<drama::Task>theTask) :
61.      TAction(theTask), _theTask(theTask) {}
62.  KickItAction::KickItAction(
63.      std::weak_ptr<drama::Task>theTask) :
64.      TAction(theTask), _theTask(theTask) {}
65.
66.  /* Define the ActionThread() methods. */
67.  void  RunAction::ActionThread(
68.      const drama::sds::Id & /*obeyArg */) {
69.
70.
71.      auto myTask(std::dynamic_pointer_cast<ClientTask>
72.                  (std::shared_ptr<drama::Task>(_theTask)));
73.
74.
75.      myTask->ServerPath().GetPath(this);
76.      myTask->ServerPath().Obey(this, "ACTION2");
```

```
77.      MessageUser("Subsidiary Action completed");
78.  }
79.
80.  void  KickItAction::ActionThread(
81.      const drama::sds::Id &kickArg) {
82.
83.
84.      GetTask()->ServerPath().Kick(this, "ACTION2", kickArg);
85.      MessageUser("Kick of Subsidiary Action completed");
86.  }
```

7.8 Changing how the methods respond to messages.

Various messages may be received when waiting for an Obey or a Kick to complete (more the Obey then the Kick). For example, trigger messages may be sent back or your action may be kicked.

The fifth parameter to the `drama::Path::Obey` and `Kick` methods specifies the address of an event processor object which is used to process these messages. This object is of class `drama::MessageEventHandler` or a subclass of it. The base class provides the most commonly required handling of messages, but you can sub-class it implement your own handling. The table below explains the methods you may wish to override. These methods will be invoked in the context of the thread that invoked the `Path::Obey` or `Kick` method, and in the DRAMA action context for the invoker, with the DRAMA lock taken. Since the DRAMA lock is taken, you must not do anything that needs wait for the main DRAMA loop to process a message, since it won't run.

Method	Description
`NewTransaction`	This method is invoked each time a message is initiated. It can be used to access the DITS transaction id of the messages, which are sent. This is useful in some cases, such as when transaction might be orphaned but will need to be handled, or, for example, if you might want to kick a *spawnable* action. The default is a null operation. The `Path::SpawnKickArg()` and `Path::SpawnKickArgUpdate()` methods can be used to convert transaction id's into arguments to be used to kick *spawnable* actions.
`ThreadWaitAbort`	Will be invoked if, whilst waiting for a message, the main DRAMA loop exits (task exiting), or the action thread completed (we must be running in a subsidiary thread) or if the user has invoked `TAction::AbortMessageWaits`.

Method	Description
	The default behavior is to throw an exception.
`TriggerReceived`	Invoked if a trigger message is received. The default reports to the user via Ers that a trigger has been received and gives a hint about how to handle it. See section 7.8.1 for an example.
`MessageRejected`	Invoked if the message sent was rejected by the target task (which happens if the action is already running or does not exist) The default behavior is to throw an exception.
`MessageComplete`	Invoked if a message complete message is received. The default behavior is to throw an exception if the status in the completion message is bad, otherwise just to allow the wait to complete.
`TaskDied`	Invoked if the subsidiary task died whilst we were waiting for a reply from it. The default behavior is to throw an appropriate exception.
`UserMessage`	Invoked if a message for the user is received. This would normally only happen for UFACE transactions (see section 8). The default behavior is to report the message to the user.
`ErrorReport`	Invoked if an error report for the user is received. This would normally only happen for UFACE transactions (see section 8). The default behavior is to report the message to the user.
`KickReceived`	Invoked if an action waiting on a message receives a kick message during the wait. The default reports to the user via Ers that a Kick has been received and gives a hint about how to handle it.

Method	Description
	Should return true to indicate the thread should continue waiting for messages from the subsidiary action, false to stop waiting and to orphan the outstanding transaction (an exception is thrown). See section 7.8.2 for more information.
`BulkTransferred`	Only received if the message send included a bulk data argument. See section 9 for details on bulk data. This message is only received if the receiver of the bulk data message is reporting progress on its use of the bulk data. The arguments provides details on the bulk data use. May be invoked zero or many times. The default is a null operation.
`BulkDone`	Only received if the message send included a bulk data argument. See section 9 for details on bulk data. Is invoked when the receiver of the bulk data message indicates it has finished reading the message, and the sender is free to reuse the item. The default is a null operation.
`WaitTimeout`	This method is invoked if one of the "...WaitUntil()" methods times out. The default implementation of this will return false. A subclass might want to override this if it is doing things that might require the timeout to change. Whilst it won't be invoked until the original timeout occurs (so you can't change to a shorter timeout), you can extend the timeout by returning true, having updated the wait until time which is supplied as an argument.

If any of these throw an exception (which many defaults do), that exception is thrown out of the call of the `Path::Obey` or `Kick` method.

Each of these methods have one or more arguments from this set of three:

- A `drama::thread::ProcessInfo` item, named `messInfo`. This contains details of the message that was sent and the sender.

- A `StatusType` argument named `status`. This provides the DRAMA status of the message. Only available if it makes sense.
- A `sds::IdPtr` parameter named `arg`. This is the SDS ID of the argument to the message, if any. Not all of the methods can have an argument.

The `drama::thread::ProcessInfo` item is the most interesting and provides access to the `drama::thread::TAction` object and `drama::Path` objects used in sending the message we are waiting for the reply to, as well as type and name of the message.

A couple of the methods have extra parameters for their special cases.

Examples are given below of handling Trigger and Kick messages, the two cases you are most likely to be interested in.

7.8.1 Responding to Trigger Messages

Example 7–6, below, shows Example 7–2 modified to handle trigger messages. The action named ACTION3 in the example_server program will send a trigger message before it completes, so that is the action sent by this example.

The class `TriggerHandler` is declared at line #1, as a sub-class of `drama::MessageEventHandler`. In this example, only the `TriggerReceived()` method is overridden, base-class defaults are used for the other methods. `TriggerReceived()` takes parameters `messInfo, status` and `arg`. This implementation ignores the status (it used only in special case trigger messages[17]), but does make use of the other method parameters.

At line #13, `messInfo.GetHandler()` is used to access the `drama::thread::TAction` object that sent the message. `GetHandler()` returns a `const` reference to a variable of type `drama::thread::TMessHandler`, of which `TAction` is a subclass. The `MessageUser()` method is then used to send a message to the user who sent the `RUN` action to this task.

Line #15 and line #17 show access to the message name and path, both via messInfo.

At line #21, the SDS argument to the trigger message is output to the user, using the `SdsListToUser()` method of `TMessHandler,` which users a sub-class of `drama::sds::PrintObjectCR` which will print the SDS Listing using `MessageUser().`

Line #44 shows an object of this type being passed to the `drama::Path::Obey()` method.

Example 7–6. Handling trigger messages

```
1.   class TriggerHandler : public drama::MessageEventHandler {
2.   public:
3.       /* Implement TriggerReceived only, others stay as defaults */
4.       void TriggerReceived(
5.           /* messInfo contains the info about the message */
6.           drama::thread::ProcessInfo messInfo,
7.           /* Message status */
8.           StatusType /*status*/,
```

[17] Trigger messages only have a non-zero status if sent by the DRAMA Parameter monitoring system. See section 7.12 for more on parameter monitoring, but the details are not required for most task authors.

```
9.          /* Any SDS arg to the message */
10.         const drama::sds::IdPtr &arg) override {
11.
12.         /* Output details of message */
13.         messInfo.GetHandler().MessageUser(
14.             "trigger message was received from action \"" +
15.             messInfo.GetMessName() +
16.             "\", to task \"" +
17.             messInfo.GetPath().GetTaskName() +
18.             "\"");
19.         /* List the argument (via MessageUser()) */
20.         if (*arg)
21.             arg->List( messInfo.GetHandler().SdsListToUser());
22.     }
23. };
24.
25. class RunAction : public drama::thread::TAction {
26.
27. public:
28.     RunAction(std::weak_ptr<drama::Task> theTask) :
29.         TAction(theTask), _theTask(theTask) {}
30.     ~RunAction() {}
31. private:
32.     std::weak_ptr<drama::Task> _theTask;
33.     void  ActionThread(const drama::sds::Id & /*obeyArg */) {
34.         drama::Path server(_theTask,
35.                            "SERVER_TASK", "",
36.                            "./example_server");
37.         server.GetPath(this);
38.         MessageUser("Have gotten path, will send obey");
39.         TriggerHandler myHandler;
40.         server.Obey(this,
41.                     "ACTION3",
42.                     drama::sds::Id::CreateNullItem(),
43.                     nullptr,
44.                     &myHandler);
45.         MessageUser("Subsidiary Action completed");
46.     }
47. };};
```

7.8.2 Responding to Kick Messages

Example 7–7, below, shows Example 7–6 modified to handle kick messages rather then trigger messages. Only the relevant sub-class of `drama::MessageEventHandler` is shown below as the rest has no significant difference from Example 7–6, please look at the source file if you want to see the rest.

This example goes back to using the action named ACTION2 in the example_server program, since that has a delay that allows us to send the Kick message. `KickReceived()` is implemented in a very similar fashion to `TriggerReceived()` above. `KickReceived()` does not have a status argument, but does return a Boolean value.

If `true` is returned, then the thread will remain blocked waiting for the subsidiary action to complete. If false is returned, then the wait is cancelled with an exception thrown. The

message transaction is orphaned (see section 7.10, below, for details on orphaned transactions).

Example 7–7. Handling kick messages whilst waiting for subsidiary message

```
1.   class KickHandler : public drama::MessageEventHandler {
2.   public:
3.       /* Implement KickReceived only, others stay as defaults */
4.       bool KickReceived(
5.           /* messInfo contains the info about the message */
6.           drama::thread::ProcessInfo messInfo,
7.           /* Any SDS arg to the message */
8.           const drama::sds::IdPtr &arg) override {
9.
10.          /* Output details of message */
11.          messInfo.GetHandler().MessageUser(
12.              "kick message was received whilst waiting for action \"" +
13.              messInfo.GetMessName() +
14.              "\", to task \"" +
15.              messInfo.GetPath().GetTaskName() +
16.              "\"");
17.          /* List the argument (via MessageUser()) */
18.          if (*arg)
19.              arg->List( messInfo.GetHandler().SdsListToUser());
20.
21.          // Return true to allow the wait to continue.
22.          // Return false to abort waiting and orphan the
23.          //  transaction.
24.          return true;
25.      }
26.  };
```

Note that when running this example, you first send the “RUN” action to CLIENT, you then kick the “RUN” action in CLIENT, e.g

```
> ./example_server &
> ./exam7_7 &
> ditscmd CLIENT RUN &
DITSCMD_6bc:CLIENT:Have gotten path, will send obey
DITSCMD_6bc:SERVER_TASK:ACTION2 invoked.
> ditscmd -k CLIENT RUN
DITSCMD_6bc:CLIENT:kick message was received whilst waiting for action
     "ACTION2", to task "SERVER_TASK"
DITSCMD_6bc:SERVER_TASK:ACTION2 complete.
DITSCMD_6bc:CLIENT:Subsidiary Action completed
```

7.8.3 Get Path messages

You can also alter how `drama::Path::GetPath()` responds to messages sent as part of loading tasks and getting paths. `GetPath()` takes an object of type `drama::thread::TransEvtProcessor` as its event processor. This is the base class of the `drama::MessageEventHandler` class used by `Obey()` and `Kick()`.

`TransEvtProcessor` only has two methods – `Process()` and `NewTransaction()`. The later works the same as the `drama::MessageEventHandler` version. `Process()` is invoked for each message received and its arguments allow access to the message details. . `Process()` will return true indicate the threads should continue waiting, false to indicate the wait should be stopped.

The default handle for GetPath() class is an object of the class `drama::GetPathEventType`.

This area of DRAMA2 is still work in progress, as we are not entirely sure of the requirements.

7.9 Message waits with timeouts

For each of the message sending and wait methods (`drama::Path::Obey()`, `Kick()`, `GetPath()`, etc.) there is an equivalent version with a timeout. These have "WaitUtill" appended to the corresponding method name: `drama::Path::ObeyWaitUntil()`, `KickWaitUntil()`, `GetPathWaitUntil()` etc. Each of these return "true" if the operation completed, false if it timed out.

In each of these, the first argument is the C++11 time point until which the method will wait. These times are specified as an absolute time using the type `std::chrono::steady_clock::time_point`. You can construct one of these items in various ways to achieve any time supported by the underlying C++11 wait on a condition variable (`std::condition_variable::wait_for()`). Much of the time you are likely to want a delay from the current time. The convenience function `drama::CreateFutureTimepoint()` provides a shortcut to this.

Example 7–8, below, shows an example.

Example 7–8. Obey with timeout.

```
1.   class RunAction : public drama::thread::TAction {
2.
3.   public:
4.       RunAction(std::weak_ptr<drama::Task> theTask) :
5.           TAction(theTask), _theTask(theTask) {}
6.       ~RunAction() {}
7.   private:
8.       std::weak_ptr<drama::Task> _theTask;
9.       void  ActionThread(const drama::sds::Id & /*obeyArg */) {
10.          drama::Path server(_theTask,
11.                             "SERVER_TASK", "",
12.                             "./example_server");
13.          server.GetPath(this);
14.          // We will wait 5 seconds.
15.          // ACTION2 waits 10 seconds, so we will timeout
16.          if (server.ObeyWaitUntil(
17.                  drama::CreateFutureTimepoint(5.0),
18.                  this, "ACTION2"))
19.          {
20.              MessageUser("Subsidiary Action completed");
21.          }
22.          else
23.          {
24.              MessageUser(
25.                  "Timeout waiting for Subsidiary Action");
26.          }
27.      }
28.  };
```

Note – when the timeout is triggered, the subsidiary action is "Orphaned". Orphans are explained below.

7.10 Orphaned Transactions

In DRAMA, if an action which has sent messages to other tasks completes before the subsidiary transactions it started have themselves completed, the subsidiary transactions are "Orphaned". As suggested, they have lost their parents. There is now nowhere to deliver any replies which come back in relation to these transactions.

In the DRAMA2 systems, transactions are orphaned if the message sending method (E.g. `drama::Path::Obey()`, `ObeyWaitUntil` etc.) throws an exception or if the WaitUntil version times out. As a result, the transaction is always orphaned if the action thread completes.

7.10.1 Default Behavior

If a message is received in relation to a transaction that has been orphaned, DRAMA will, by default, print information about it to `stderr`.[18] For example, Example 7–8, above will project something like this:

```
CLIENT:SERVER_TASK:ACTION2 invoked.
##CLIENT:Orphan Transaction Completed (DRAMA2 OrphanHandler).
# CLIENT:Orphan Message = Reason for entry = informational message
received.
# CLIENT:Task::OrphanHandler:see log file for details
##CLIENT:Orphan Transaction Completed (DRAMA2 OrphanHandler).
# CLIENT:Orphan Message = Reason for entry = completion message received,
status = OK.
# CLIENT:Orphaned completion of action "ACTION2" received from task
"SERVER_TASK".
# CLIENT:Task::OrphanHandler:see log file for details
```

In this example, two messages have been received, an informational message (`MessageUser()/MsgOut()`) and an action completion message. The `MsgOut()` message is output (the first line) before a message about the orphan is output. For the later, the action name and task are also output. In both cases, if the task is logging (section 11) then more information will be written to the log file.

7.10.2 Changing the default behavior

An implementation may prefer that the task handle any orphans itself, rather then the default of messages to `stderr`. This can be done by overriding the method `Task::OrphanHandler(const OrphanDetails &details)` in your DRAMA task implementation. The `details` argument provides access to information about the transaction.

This method is invoked by the thread running `drama::task::RunDrama()` after processing a DRAMA message but before blocking to wait for a message. The DRAMA lock is taken so you can't do anything that waits for a message.

[18] Actually, the details are output using ERS and `MsgOut()`. By default ERS messages sent outside an action are output to `stderr`, but the `DitsUfacePutErsOut()` routine can change what happens. Similarly, `MsgOut()` messages sent outside action are output to `stdout`, but `DitsUfacePutMsgOut()` can change that.

7.10.3 Actions taking over orphans

In traditional DRAMA tasks, an action may adopt orphans. This is done using the C routine `DitsTakeOrphans()`. This might be done by say a POLL action that is normally started by a GUI task. The POLL action can then ensure details are reported back to the user via the GUI task.

Whilst this can be done in DRAMA2, it is not *yet* possible for a threaded action to take over orphans. All dealing with such transactions must be handled using the traditional C interfaces to DRAMA.

Additionally, there are some orphans for which this does not work. An action thread may receive a series of messages in quick succession. In this case, the thread running `drama::task::RunDrama()` may queue the handling of the events to the action thread but not have an opportunity to yield the CPU to the action thread. When it finally does yield to the action thread, one of the events might cause the action thread to terminate its wait, before all events are processed. The result is that any already queued events for that action become orphan transactions. Since this is occurring after the message has already been delivered and processed by `drama::task::RunDrama()`, the events cannot be delivered to another action. DRAMA2 tries had to avoid this case (it will try to yield the CPU after each messages) but there is no certainty here at this point since it is dependent on the actual OS thread scheduling policy.

7.10.4 Creating an orphan on demand.

There are cases where you may wish to create a transaction without a parent – sending an message you have no interest in seeing the reply of. This is accomplished by using one of the `…WaitUntil()` set of methods to send the message, having set the time point to some point in the past. If you do this, the transaction will be orphaned immediately.

7.11 Parameter Set/Get Messages

You can send messages to set and get the values of parameters in other tasks. The methods `drama::Path::SetParam()` and `drama::Path::GetParam()` are used. Compared to say `drama::Path::Obey()`, these are a little simpler – if you are setting a parameter where is no output argument, whist if you are getting the value, there is no input argument. There is also a version of `GetParam()` that takes a list of parameters to return. Example 7–9, below, shows them in action. At line 20, it is getting the value of the parameter named “PARAM1”. This is then set to a new value at line 32. Then at line 35 the list version of `GetParam()` is used to return the values of 3 parameters in one operation.

Example 7–9. Getting and setting parameters

```
1.   class RunAction : public drama::thread::TAction {
2.
3.   public:
4.       RunAction(std::weak_ptr<drama::Task> theTask) :
5.           TAction(theTask), _theTask(theTask) {}
6.       ~RunAction() {}
7.   private:
8.       std::weak_ptr<drama::Task> _theTask;
9.       void  ActionThread(const drama::sds::Id & /*obeyArg */) {
10.
11.          drama::Path server(_theTask,
12.                             "SERVER_TASK", "",
13.                             "./example_server");
```

```
14.          server.GetPath(this);
15.
16.          // Used to store results of GetParam messages
17.          drama::sds::IdPtr val;
18.
19.         // Get PARAM1 back and list it.
20.          server.GetParam(this, "PARAM1", &val);
21.          MessageUser("Get PARAM1 complete:");
22.          val->List(SdsListToUser());
23.
24.          // Grab the value of the PARAM1.
25.          INT32 ival;
26.          val->Get("PARAM1", &ival);
27.
28.          // Set SERVER_TASK PARAM1.
29.          drama::sds::Id newVal =
30.              drama::sds::Id::CreateArgStruct();
31.          newVal.Put("Argument1", ival+2);
32.          server.SetParam(this, "PARAM1", newVal);
33.          MessageUser("Set PARAM1 complete");
34.          // Get of a list of parameters and list.
35.          server.GetParam(this,
36.                          {"PARAM1", "PARAM2", "PARAM3"},
37.                          &val);
38.
39.          MessageUser("Get list of parameters complete:");
40.          val->List(SdsListToUser());
41.
42.      }
43.  };
```

7.12 Parameter Monitoring

Parameter Monitoring is one of DRAMA's most powerful features. It allows a task to make its state, as provided for in its parameters, available to other tasks which get updates as things change in a very efficient way (no polling). For example, a server task running a motor may maintain the position of the motor in a parameter. A GUI can "monitor" that parameter and display the value to the user. The application code in the server task does not need to do anything other then put the value into a parameter and tell DRAMA it has been updated. In most cases the later is done by the function updating the parameter value[19].

There can be several tasks monitoring the server's parameters at the same time, again, the server task application code does not need to do anything to enable this – in fact, it does not normally know this is happening. It has made its state public via parameters and does not care what the public does from that point.

When an update to a parameter value is made, its value is sent to all client tasks that have expressed an interest in it (monitored it). Updates are only sent when updates occur – no polling is needed by any task involved.

It is possible for the performance of server task to be impacted (as it is sending more messages), but this has rarely been seen in practice. The most likely cause is updating parameters at a very high rate – triggering a large number of messages to be sent. From a GUI

[19] If updating a complex SDS parameter or updating parameter values by pointer – you will probably need to tell DRAMA you have done the update. See 4.6 for more information. Also the C function `SdpUpdate()`.

point of view, 3 times a second is normally fast enough. Flooding a GUI with updates triggering screen changes will likely cause the GUI performance to degrade well before the server performance degrades.

7.12.1 Standard monitoring vs. forward monitoring

There are two types of monitoring. The first, which we are referring to here as "Standard Monitor", is where a task requesting a monitor wants the results sent to it. This is the most common type of monitor.

The second type of monitor is the "Forward Monitor". Whilst only used in some cases, it is a powerful technique. The task requesting the monitor requests that changes are sent to a third task, causing an action to be invoked in that task with the new parameter value as the action argument.

Why would you want a forward monitor? They provide for efficient designs and avoid complex dependency issues. For example, consider a system with three tasks. There is a camera server task – it runs a camera and reads images from it. There is a standard image display task. It knows how to display an image, but knows nothing about the camera server itself. And there is a GUI task that controls both.

The effect I am looking for is that every image produced by the camera server is displayed on the image display task for the user. I want to implement the camera server using a standard DRAMA interface, so I can later replace that server with another for a different camera. And I want the flexibility to replace the image display task with an alterative in the future. Additionally, the camera server should still work even if the image display is not running. So neither the camera server nor the image display task should have a dependency on each other.

To make the camera server work regardless of if the image display is connected, I can use a parameter to store the result of each read of the camera. That way, any client can grab the last image taken at will. And that naturally leads to parameter monitoring as the way to get updates every time the camera is read.

With standard monitor, the design would be that the GUI monitors the parameter, and when it gets an update, it forwards the parameter onto an action in the image display task. But the GUI may have no interest in the image itself, in which case this double handling is annoying. If large messages are involved (in this case, they are images so could be MBs in size) and particularly if the tasks in question are distributed across multiple machines, this double handling is particularly inefficient.

With "Forward Monitoring", the GUI can tell the camera server task to forward the value of the parameter directly to an action in the image display task. As long as the parameter value format is accepted by the action in the image display, this will work. This design is much neater. All the details are hidden by DRAMA and double handling is avoided.

7.12.2 The Monitor Messages.

The messages to start monitors are a DRAMA message type in the same way that Obey, Kick, Set Parameter and Get Parameter messages are. But as these are handled internally to DRAMA, there are limited possibilities. There are four sub-types and each has an associated method in the `drama::Path` class.

Monitor Message Sub-Type	`drama::Path` member	Description

Monitor Message Sub-Type	`drama::Path` member	Description
Start	`MonitorStart()`	Starts standard monitoring of all parameters the names of which are specified in the argument. As noted below, avoid using this, use a subclass of `thread::Monitor` to run these.
Forward	`MonitorForward()`	Various overloads of this method allow specification of the task to which the parameter is forwarded, the action it is forwarded with and the names of the parameters to be monitored.
Cancel	`MonitorCancel()`	Cancel a monitor operation. This causes the `MonitorStart()` or `MonitorForward()` to complete.
Add	TBD	Add a specified parameter from the set being monitored for a particularly monitor message
Delete	TBD	Delete a specified parameter from the set being monitored

The Add and Delete message types are not yet supported, as these have never in practice been used. They would be easy to add if needed.

In reality, you should NOT used the `MonitorStart()` call. Instead you should use a sub-class of `drama::thread::Monitor`. This class wraps up most of the work involved. You create one of them with the list of parameters to monitor supplied to the contractor. You then invoke the `Monitor::Run()` method on the object. The `drama::thread::Monitor::ParameterChanged()` method will be invoked on each parameter change. Your sub-class implements this method to respond to monitor events.

Some subclasses of `drama::thread::Monitor` have been implemented for the most common cases – copying values into a tasks own parameters and implementing a basic response vetted by type.

7.12.3 Monitor to Parameters

The `drama::thread::MonitorToParam` class is a sub-class `thread::Monitor` of designed to copy the values of given sub-class parameters directly into parameters of the task self. This is a concrete sub-class, so can be used as it, but in many cases you will want to override the `Transform()` method. This method is invoked to allow you store the value in a parameter of a different name from the name used in the task being monitored.

Example 7–10, below, shows this being done. Here the RUN action, from line 34, will monitor parameters `PARAM1`, `PARAM2` and `PARAM3` in the example server task, with the

monitor constructed at line 45 and run at line 52. During task initialization, equivalent parameters were created in this task prefixed by "T_".

A sub-class of `drama::thread::MonitorToParam` is used – line #9. This overrides the `Transform()` method to prefix the name of the parameter being monitored to the name of the parameter in our task. `Transform()` is invoked each time a monitor event happens – each time the parameter changes. Note that you will probably want to "use" the constructor from `MonitorToParam`, as per the `using` specification on line 11. That avoids implementing it yourself.

Example 7–10. Monitoring to parameters

```
1.  /*
2.   * Create a class used to implement our monitors.  This is a
3.   * sub-class of MonitorToParam which will update parameters
4.   * in our task with the values of monitored parameters.  We
5.   * subclass MonitorToParam so that we can transform the name
6.   * of the parameter being monitored to the name of
7.   * the parameter we want updated.
8.   */
9.  class MyMonitor : public drama::thread::MonitorToParam {
10.     // Use the MonitorByType constructors.
11.     using  drama::thread::MonitorToParam::MonitorToParam;
12.     // Override Transform.
13.     std::string Transform(const std::string &in) override;
14. };
15. /*
16.  * Invoked to transform the name of a parameter being
17.  * monitored to the name of a parameter in this task
18.  * that we want the value copied to.
19.  */
20. std::string MyMonitor::Transform(const std::string &in)
21. {
22.     // Just add the T_ prefix to the supplied name.
23.     return std::string("T_") + in;
24. }
25.
26. class RunAction : public drama::thread::TAction {
27.
28. public:
29.     RunAction(std::weak_ptr<drama::Task> theTask) :
30.         TAction(theTask), _theTask(theTask) {}
31.     ~RunAction() {}
32. private:
33.     std::weak_ptr<drama::Task> _theTask;
34.     void  ActionThread(const drama::sds::Id & /*obeyArg */) {
35.
36.         drama::Path server(_theTask,
37.                            "SERVER_TASK", "",
38.                            "./example_server");
39.         server.GetPath(this);
40.         /*
41.          * Create a monitor object, specifying a list
42.          * of parameters in the SERVER_TASK task to
43.          * be monitored.
44.          */
45.         MyMonitor myMonitor(
46.             _theTask,
47.             { "PARAM1" , "PARAM2", "PARAM3" });
48.
```

```
49.          try
50.          {
51.              MessageUser("About to run monitor");
52.              myMonitor.Run(&server, this);
53.          }
54.          catch (drama::Exception &e)
55.          {
56.              /*
57.               * If the above threw a DRAMA Run loop exit
58.               * error, then this task was shutdown whilst
59.               * we were running the monitor. In this case,
60.               * we are not worried and allow the thread
61.               * to close down without error messages.
62.               * But for any other reason, we re-throw
63.               * the exception.
64.               */
65.              if (e.dramaStatus() != DRAMA2__RUN_LOOP_EXIT)
66.              {
67.                  throw e;
68.              }
69.          }
70.
71.      }
72.  };
73.
74.  class ClientTask : public drama::Task {
75.
76.  private:
77.      RunAction RunActionObj;
78.      drama::Parameter<int>         param1;
79.      drama::Parameter<std::string> param2;
80.      drama::Parameter<float>       param3;
81.  public:
82.      /**
83.       * Constructor, from here we add actions
84.       */
85.      ClientTask(const std::string &taskName) :
86.          drama::Task(taskName), RunActionObj(TaskPtr()),
87.          param1(TaskPtr(), "T_PARAM1", 0),
88.          param2(TaskPtr(), "T_PARAM2", ""),
89.          param3(TaskPtr(), "T_PARAM3", 0.0) {
90.
91.          Add("RUN", drama::MessageHandlerPtr(
92.                  &RunActionObj, drama::nodel()));
93.          Add("EXIT", drama::SimpleExitAction);
94.      }
95.      ~ClientTask() {
96.      }
97.  };
```

Running Example 7–10 is a bit complicated. You need to use "`ditscmd -g`" commands to check parameters and "ditscmd –s" to change parameter values. See the log below for an example of working with this.

```
>> ./example_server&
[2] 9138
>> ./exam7_10 &
[4] 9169
```

```
>>
>> ditscmd CLIENT RUN &
[5] 9171
>> DITSCMD_23d4:CLIENT:About to run monitor
>>
>> ditscmd -g SERVER_TASK _ALL_
DITSCMD_23dd:SdpStructure          Struct
DITSCMD_23dd:  LOG_LEVEL             Char    [5] "NONE"
DITSCMD_23dd:  GITLOG_FILENAME       Char    [1] ""
DITSCMD_23dd:  PARAM1                Int      2
DITSCMD_23dd:  PARAM2                Char    [13] "hi there c++"
DITSCMD_23dd:  PARAM3                Float    33.3
DITSCMD_23dd:  PARAM4                UInt     4
>> ditscmd -g CLIENT _ALL_
DITSCMD_23ec:SdpStructure          Struct
DITSCMD_23ec:  LOG_LEVEL             Char    [5] "NONE"
DITSCMD_23ec:  GITLOG_FILENAME       Char    [1] ""
DITSCMD_23ec:  T_PARAM1              Int      2
DITSCMD_23ec:  T_PARAM2              Char    [13] "hi there c++"
DITSCMD_23ec:  T_PARAM3              Float    33.3
>> ditscmd -s SERVER_TASK PARAM1 22
>> ditscmd -s SERVER_TASK PARAM2 "does monitoring work"
>> ditscmd -s SERVER_TASK PARAM3 3333.3
>> ditscmd -g CLIENT _ALL_
DITSCMD_2431:SdpStructure          Struct
DITSCMD_2431:  LOG_LEVEL             Char    [5] "NONE"
DITSCMD_2431:  GITLOG_FILENAME       Char    [1] ""
DITSCMD_2431:  T_PARAM1              Int      22
DITSCMD_2431:  T_PARAM2              Char    [21] "does monitoring work"
DITSCMD_2431:  T_PARAM3              Float    3333.3
>> ditscmd CLIENT EXIT
```

7.12.4 Monitor by Type

When monitoring, one of the first things often done is to break parameters being monitored up via the parameter type. The `drama::thread::MonitorByType` sub-class of `thread::Monitor` does this, providing separate `ParamChanged()` methods to be invoked for each basic type. The sub-class can override these as required.

Example 7–11 below, is similar to Example 7–10 but is using `MonitorByType`. The `MyMonitor` class overrides various of the `ParamChanged()` methods of `MonitorByType`. After working through Example 7–10, I think the reader will find this easy to understand. The example just outputs the changed parameter values to `stderr`.

Example 7–11. Monitoring by type

```
1.   /*
2.    * Create a class used to implement our monitors.  This is a
3.    * sub-class of MonitorToParam which will update parameters
4.    * in our task with the values of monitored parameters.  We
5.    * subclass MonitorToParam so that we can transform the name
6.    * of the parameter being monitored to the name of
7.    * the parameter we want updated.
8.    */
9.   class MyMonitor : public drama::thread::MonitorByType {
10.      // Use the MonitorByType constructors.
11.      using  drama::thread::MonitorByType::MonitorByType;
12.      /*
13.       * Each of these override methods in MonitorByType.
```

```
14.       *  Mark them as "override" to ensure we have
15.       *  the signature correct.
16.       */
17.
18.      /*
19.       * Invoked when a signed integer parameter is changed.
20.       */
21.      void ParamChanged(const std::string &name,
22.                        long value) override;
23.      /*
24.       * Invoked when a string parameter is changed, also
25.       * for any of int/unsigned int/real which we have
26.       * not overriden, so in this case, real/unsigned int
27.       * cases as well.
28.       */
29.      void ParamChanged(const std::string &name,
30.                        const std::string & value) override;
31.      /*
32.       * Invoked when some complex parameter (array of
33.       * items or structured item) is changed.
34.       */
35.      void ParamChanged(const std::string &name,
36.                        const drama::sds::IdPtr &value)
37.          override;
38.
39.  };
40.  /*
41.   * Invoked when an integer parameter being monitored changed.
42.   */
43.  void MyMonitor::ParamChanged(const std::string &name,
44.                               long value)
45.  {
46.      std::cerr << "*** MyMonitor:ParamChanged (int) \""
47.                << name
48.                << "\" to \""
49.                << value
50.                << "\""
51.                << std::endl;
52.  }
53.
54.  /*
55.   * Invoked when a string parameter being monitored changed.
56.   *  Also invoked for any of int/unsigned/real which we don't
57.   *  explicitly support.
58.   */
59.  void MyMonitor::ParamChanged(
60.      const std::string &name,
61.      const std::string & value)
62.  {
63.
64.      std::cerr << "*** MyMonitor:ParamChanged (string) \""
65.                << name
66.                << "\" to \""
67.                << value
68.                << "\""
69.                << std::endl;
70.  }
71.
72.  /*
73.   * Invoked when a complex parameter changes.
74.   */
```

```
75.  void MyMonitor::ParamChanged(
76.      const std::string &name,
77.      const drama::sds::IdPtr &value)
78.  {
79.      std::cerr << "*** MyMonitor:ParamChanged (complex) \""
80.                << name
81.                << "\""
82.                << std::endl;
83.
84.
85.      value->List(std::cerr);
86.  }
87.  class RunAction : public drama::thread::TAction {
88.
89.  public:
90.      RunAction(std::weak_ptr<drama::Task> theTask) :
91.          TAction(theTask), _theTask(theTask) {}
92.      ~RunAction() {}
93.  private:
94.      std::weak_ptr<drama::Task> _theTask;
95.      void  ActionThread(const drama::sds::Id & /*obeyArg */) {
96.
97.          drama::Path server(_theTask,
98.                             "SERVER_TASK", "",
99.                             "./example_server");
100.         server.GetPath(this);
101.         /*
102.          * Create a monitor object, specifying a list
103.          * of parameters in the SERVER_TASK task to
104.          * be monitored.
105.          */
106.         MyMonitor myMonitor(_theTask,
107.                             { "PARAM1" , "PARAM2",
108.                               "PARAM3", "PARAM4" ,
109.                               "STRUCT_PARAM" });
110.
111.         try
112.         {
113.             MessageUser("About to run monitor");
114.             myMonitor.Run(&server, this);
115.         }
116.         catch (drama::Exception &e)
117.         {
118.             /*
119.              * If the above threw a DRAMA Run loop exit
120.              * error, then this task was shutdown whilst
121.              * we were running the monitor. In this case,
122.              * we are not worried and allow the thread
123.              * to close down without error messages.
124.              * But for any other reason, we re-throw
125.              * the exception.
126.              */
127.             if (e.dramaStatus() != DRAMA2__RUN_LOOP_EXIT)
128.             {
129.                 throw e;
130.             }
131.         }
132.
133.     }
134. };
135.
```

```
136. class ClientTask : public drama::Task {
137.
138. private:
139.     RunAction RunActionObj;
140. public:
141.     /**
142.      * Constructor, from here we add actions
143.      */
144.     ClientTask(const std::string &taskName) :
145.         drama::Task(taskName), RunActionObj(TaskPtr()) {
146.
147.         Add("RUN", drama::MessageHandlerPtr(
148.                 &RunActionObj, drama::nodel()));
149.         Add("EXIT", drama::SimpleExitAction);
150.     }
151.     ~ClientTask() {
152.     }
153. };};
```

Below see an example of running Example 7–11.

```
>> ./example_server&
[2] 27148
>> ./exam7_11&
[4] 27149
>>
>> ditscmd CLIENT RUN &
[5] 27150
>> DITSCMD_6a0f:CLIENT:About to run monitor
*** MyMonitor:ParamChanged (int) "PARAM1" to "2"
*** MyMonitor:ParamChanged (string) "PARAM2" to "hi there c++"
*** MyMonitor:ParamChanged (string) "PARAM3" to "33.3"
*** MyMonitor:ParamChanged (string) "PARAM4" to "4"
*** MyMonitor:ParamChanged (complex) "STRUCT_PARAM"
STRUCT_PARAM          Struct
  STRUCT_VALUE_1        Int     11
  STRUCT_VALUE_2        Int     21

>> ditscmd -s SERVER_TASK PARAM1 22
*** MyMonitor:ParamChanged (int) "PARAM1" to "22"
>> ditscmd -s SERVER_TASK PARAM2 "I am go"
*** MyMonitor:ParamChanged (string) "PARAM2" to "I am go"
>> ditscmd -s SERVER_TASK PARAM3 444.4
*** MyMonitor:ParamChanged (string) "PARAM3" to "444.4"
>> ditscmd -s SERVER_TASK STRUCT_PARAM.STRUCT_VALUE_1 343
*** MyMonitor:ParamChanged (complex) "STRUCT_PARAM"
STRUCT_PARAM          Struct
  STRUCT_VALUE_1        Int     343
  STRUCT_VALUE_2        Int     21
>> ditscmd CLIENT EXIT
>> DITSCMD_6a0f:exit status:%DITS-F-TASKDISC, Task disconnected

[5]  + exit 4     ditscmd CLIENT RUN
>>
[4]  + done       ./exam7_11
```

7.12.5 Forward Monitors.

Implement these viaone of the `drama::Path::MonitorForward()` methods. The task specified is the task too which the value of the parameter is to be sent, the action is the action it will be sent as an argument too.

7.12.6 Cancelling Monitors

The `drama::thread::Monitor` class provides the `Cancel()` method to cancel a monitor. Invoking this will cause a new thread to be created which sends the appropriate message to cancel the monitor. The `drama::thread::Monitor::Run()` will then return.

When invoked from a threaded action, `drama::thread::Monitor::Run()` will also respond to DRAMA Kick messages. The default behavior is to cancel the monitor, but the user can subclass the `drama::thread::MonitorMessageHandler`, overriding `KickReceived()`, to perform as required.

7.13 Control Messages

The final message type is the *Control* message. These were implemented as a way of sending messages to the DRAMA component of a task, rather then the application part. These are, naturally, sent using `drama::Path::Control()`. They use of this method is exactly the same as per `Path::Obey()`. As a result, non example is provided. The "`-c`" option to "`ditscmd`" allows you to send control messages. The table below details the currently supported control messages.

Name	Argument	Description
`DEFAULT`	Option. Directory Spec	If an argument is supplied, it is a new default directory. Otherwise the default directory is returned
`MESSAGE`	Status Code	The argument is a DRAMA Status Code. Convert it to its string representation. Allows an external program to translate codes known to the target task.
`DEBUG`	Integer debug level	Set the DRAMA Internal debugging level. Set $DITS_DIR/DitsSystem.h for details of the codes.
`DUMPPATHS`	None	Dump details of known paths.
`DUMPTRANSIDS`	None	Dump details of outstanding transactions
`DUMPACTACTIVE`	None	Dump details of active actions
`DUMPACTALL`	None	Dump details of all actions
`DUMPMON`	None	Dump details of running monitors
`VERSIONS`	None	Dump DRAMA, DITS and IMP version numbers

LOGNOTE	Some text	Write the argument (a string) to the log file, if there is one.
LOGFLUSH	None	Flush the log file (if supported by the logger)
LOGINFO	None	Output details of the logger (if supported by the system)
SDSLEAKCHK	None	Output details to help track SDS ID leaks.
HELP	None	Output the list of messages.
PRINTENV	Env var name	Translate an environment variable. If no argument, then output all.

There are, as yet, no DRAMA 2 specific messages. They may be added.

7.14 Multiple simultaneous messages from one action.

There is not much fun to be had if you can only send one simultaneous message from your action. In traditional (C language) DRAMA an action have as many messages outstanding simultaneously as resources allow. For example, the AAO's 2dF Control task will initialize/reset all of the 12 or so tasks it controls directly simultaneously. Since many of these run on other machines, the initialization can run in parallel and the startup of the entire system is limited only by the initialization time of the slowest thing to start up. But the 2dF Control task must reschedule to handle replies for all of these and puts significant effort into managing them so that it knows they are done and any errors are reported.

As used so far, DRAMA 2 messaging blocks the action thread whilst sending one message and you cannot be simultaneously sending others. The approach used in DRAMA 2 to support multiple outstanding messages is to use a thread for each message. The action code must start a new thread for each message it wants to be sent simultaneously. Each thread can block as required. The application code is responsible for joining the threads back into the action thread itself.

Consider Example 7–11 above. In this example, you can kick the action running the monitor to cancel the monitor. Support for this is implemented by the `drama::thread::MonitorByType` class. In DRAMA, to cancel a monitor, you need to send a MONTOR CANCEL message to the task being monitored, specifying an ID returned when the monitor is started. So to do this, the action needs to have two messages outstanding at the same time. The `MonitorByType` implements the cancel by creating a thread that sends the MONITOR CANCEL messages.

A simple example of this approach is given by Example 7–12, below. This work was started from Example 7–2. But instead of sending obey message (`ACTION1`) to the server task, we are sending 4 of them simultaneously. We will send `ACTION1`, `ACTION2`, `ACTION3` and `ACTION4`.

First a bit of house keeping; some of these actions send trigger messages back, which we are not interested in. So we sub-class `drama::MessageEventHandler` to create the class `TriggerIgnorer`, and use an object of that class to handle messages and just ignore the triggers. See lines 5 to 16

We will create a child thread for each message we want to send, and then have the original action thread wait on futures to be set when each child thread completes.

We can, in this simple example, use the same function to run each thread. This is the function `SendObey()`, from line 19. The only thing you haven't see before is the call to `drama::thread::BlockSignals()`. This function will block all blockable signals from being delivered to this thread. This is done because Unix signals do not work well with Unit threads. We prefer that only the thread running the DRAMA loop receive any signals. Otherwise, this we create the event handler object and send the obey. The thread completes when the Obey has completed.

Back in the action thread itself, instead of sending the obey ourselves (as in Example 7–2) we need to start a thread for each action we will send. The simplest approach is to use `std::async()`, see line 55. This returns a `std::future` we can use to wait for the thread to complete.

The most annoying thing about a `std::future` is that there is (as of C++14) no way of waiting the first of a set of futures to complete. You can only wait on one at a time. So in the wait code, lines 63 to 66, the example just waits on each in sequences. This works, but would be annoying in some cases. Resolution left to the reader, if needed.

Note that if one the child threads throws an example, then this will be transferred to the action thread at the corresponding `std::future::get()` operation.

Example 7–12. Sending multiple messages from one action thread

```
1.  /*
2.   * Implement our own event handler,
3.   *   which ignores trigger messages
4.   */
5.  class TriggerIgnorer : public drama::MessageEventHandler {
6.  private:
7.      std::weak_ptr<drama::Task> _theTask;
8.  public:
9.      TriggerIgnorer(std::weak_ptr<drama::Task> theTask) :
10.         _theTask(theTask) {}
11.     void TriggerReceived(
12.         drama::thread::ProcessInfo /*messInfo*/,
13.         StatusType /*status*/,
14.         const drama::sds::IdPtr &/*arg*/) override {
15.     }
16. };
17.
18. /* This is executed in a thread */
19. void SendObey(
20.     std::weak_ptr<drama::Task> theTask,
21.     drama::Path *path,
22.     drama::thread::TMessHandler *action,
23.     std::string obeyActionName)
24. {
25.     drama::thread::BlockSignals();
26.     TriggerIgnorer triggerIgnore(theTask);
27.     path->Obey(action, obeyActionName,
28.                drama::sds::Id::CreateNullItem(),
29.                nullptr,
30.                &triggerIgnore);
31. }
32.
33. class RunAction : public drama::thread::TAction {
```

```
34.
35.  public:
36.      RunAction(std::weak_ptr<drama::Task> theTask) :
37.          TAction(theTask), _theTask(theTask) {}
38.      ~RunAction() {}
39.  private:
40.      std::weak_ptr<drama::Task> _theTask;
41.      void  ActionThread(const drama::sds::Id & /*obeyArg */) {
42.          drama::Path server(_theTask,
43.                             "SERVER_TASK", "",
44.                             "./example_server");
45.
46.          const unsigned numActions = 4;
47.          std::vector<std::future<void>> futures(numActions);
48.
49.          server.GetPath(this);
50.
51.          for (unsigned i = 0; i < numActions ; ++i)
52.          {
53.              std::string actName = "ACTION" +
54.                  std::to_string(i+1);
55.              futures[i] = std::async(std::launch::async,
56.                                      SendObey,
57.                                      _theTask,
58.                                      &server,
59.                                      this,
60.                                      actName);
61.          }
62.          MessageUser("Waiting for subsidiary actions");
63.          for (unsigned i = 0; i < numActions ; ++i)
64.          {
65.              futures[i].get();
66.          }
67.          MessageUser("All actions complete");
68.      }
69.  };};
```

Earlier on, this section was discussing the implementation of some of the code behind Example 7–11. This is a useful example as it demonstrates the sending messages from Kick handlers, and you may actually want to re-implement such code to handle the kicks differently (possibly using an argument to work out what you really need to do.

We will re-implement Example 7–11 to explicitly code the handling of the kick message to cancel the monitor. Example 7–13 through to Example 7–16 are all from the one source file, exam7_13. It is a relatively complex file, so we will just look at the significant bits. You can look at the example source for the full details.

The first thing we need to do is to intercept the Kick of the RUN action when the Monitor is running. We need a sub-class of `drama::thread::MonitorMessageHandler`, which itself a sub-class of `drama::MessageEventHandler`. Example 7–13 shows the new class declaration – `MyMonitorMessageHandler`. We implement the constructor so we can grab some information we need and the `KickReceived` method.

Example 7–13. Monitor Kick Handler - MonitorMessageHandler

```
1.   class MyMonitorMessageHandler :
2.   public drama::thread::MonitorMessageHandler {
3.   private :
4.       // Keep information KickReceived will need.
```

```
5.      drama::thread::Monitor &_theMonitor;
6.      drama::Path *_thePath;
7.      drama::thread::TMessHandler *_action;
8.
9.  public:
10.     MyMonitorMessageHandler(
11.         drama::thread::Monitor &theMon,
12.         drama::Path *path,
13.         drama::thread::TMessHandler *action) :
14.
15.         drama::thread::MonitorMessageHandler(theMon),
16.         _theMonitor(theMon),
17.         _thePath(path),
18.         _action(action) {}
19.     bool KickReceived(drama::thread::ProcessInfo messInfo,
20.                       const drama::sds::IdPtr &arg) override;
21. };
```

In the `ActionThread()` method, we construct one of these objects and pass it to the monitor `Run()` method. As per Example 7–14 below, lines 9 and 12.

Example 7–14 Monitor Kick Handler – Action Code.

```
1.  void  ActionThread(const drama::sds::Id & /*obeyArg */) {
2.  …
3.          try
4.          {
5.              MessageUser("About to run monitor");
6.              /*
7.               * Create my own message handler
8.               */
9.              MyMonitorMessageHandler messHandler(
10.                 myMonitor, &server, this);
11.             canceling = false;
12.             myMonitor.Run(&server, this, &messHandler);
13.
14.             /*
15.              * If we canceled, need to wait on the
16.              * future to force the thread to join.
17.              */
18.             if (canceling)
19.             {
20.                 MessageUser("Monitor was canceled");
21.                 cancelFuture.get();
22.             }
23.
24.         }
25.         catch (drama::Exception &e)
```

The MyMonitorMessageHandler::KickReceived method is the next thing to consider, as shown in Example 7–15. It is actually very simple. First it checks that the monitor is actually running using (`IsRunning()` true and `GetMonitorId() >= 0`), Note that `IsRunning()` might be true and (`GetMonitorId() < 0`) if the operation is just starting. Unfortunately there is no way of cancelling the monitor in this state, you need to wait until it is started. Your application might need to consider that case in more detail.

It also does not want to send a second monitor cancel operation, so the "`cancelling`" flag is used to check that. If we were to send a second cancel, it would likely fail and we would have to manage multiple threads sending cancels – so in this example, we don't bother.

Then we just start the thread, running the `SendCancel()` function. The future returned by `std::async()` is stored.

Example 7–15. Monitor Kick Handler – processing the kick.

```
1.   bool  MyMonitorMessageHandler::KickReceived(
2.       drama::thread::ProcessInfo /*messInfo*/,
3.       const drama::sds::IdPtr & /*arg*/)
4.   {
5.       // Confirm the monitor is running.
6.       if ((_theMonitor.IsRunning())&&
7.           (_theMonitor.GetMonitorId() >= 0)&&
8.           (!canceling))
9.       {
10.          /*
11.           * Indicate we are canceling monitoring.
12.           */
13.
14.          canceling = true;
15.          /*
16.           * Start the thread
17.           */
18.          cancelFuture = std::async(std::launch::async,
19.                                    SendCancel,_thePath, _action,
20.                                    _theMonitor.GetMonitorId());
21.
22.      }
23.      /*
24.       * Returning true says the wait of the
25.       * message to complete should continue.
26.       */
27.      return true;
28.
29.  }
```

The implementation of `SendCancel()`, Example 7–16, is very similar to SendObey() in Example 7–12, above. It is simplified by not having to worry about handling trigger messages.

Example 7–16 Monitor Kick Handler –sending the Monitor Kick.

```
1.   void SendCancel(drama::Path *path,
2.                   drama::thread::TMessHandler *action,
3.                   int monitorId)
4.   {
5.       // First block all signals.
6.       drama::thread::BlockSignals();
7.       path->MonitorCancel(action, monitorId);
8.       // At this point, the cancel message has completed.
9.   }
```

Finally, back in the `ActionThread()` method in Example 7–14, we must deal with having sent a cancel. After the `myMonitor.Run()` method returns, we must check if we were being cancelled and if true, do a `get()` on the future.

Note that correct exception handling potentially complicated, as `myMonitor.Run()` and the `get()` may both throw exceptions. If `myMonitor.Run()` throws an exception and you are canceling, you will probably want to do the `get()` on the future (to ensure resources are tided up), ignore any exception it would throw and re-throw what `myMonitor.Run()` threw. But if `myMonitor.Run()` returned without an exception and the `get()` throws, you probably want that to be thrown by `ActionThread()`. This implementation is left for the reader.

7.15 Sequencing issues.

The DRAMA2 implementation, due to its use of threads, experiences sequencing issues not seen in traditional DRAMA tasks. In particular, the DRAMA message reading loop may read a number of messages before the threads that must be woken are run.

Consider a task named CLIENT with an action named RUN. Presume the RUN action will send message C1 to the user, and obey the action WORK in the task SERVER. It might get a trigger message from WORK in SERVER, at which point it sends message C2 to the user. When WORK in SERVER completes, it sends Message C3 to the user.

Then presume the SERVER task WORK action has a simple sequence. It sends message S1, sends a trigger message and then sends message S2 before completing.

The order of messages for a traditional C implementation of CLIENT is always:

- Message C1 (CLIENT RUN action starting)
- Message S1 (SERVER WORK action starting)
- Message C2 (CLIENT response to Trigger)
- Message S2 (SERVER WORK action about to end)
- Message C3 (CLIENT RUN action complete)

But if you implement CLIENT with DRAMA2, you may, in some cases get the following sequence:

- Message C1 (CLIENT RUN action starting)
- Message S1 (SERVER WORK action starting)
- Message S2 (SERVER WORK action about to end)
- Message C2 (CLIENT response to Trigger)
- Message C3 (CLIENT RUN action complete)

That is, the CLIENT task appears to get the Trigger message after the SERVER action has completed.

This happens because messages to users sent from a subsidiary action are not handled by the action thread. They are dealt with by DRAMA without the action thread being invoked. In the above example, message S1 was received from the client and sent immediately to the user. The trigger message was received, but used to schedule a wake-up of the RUN action thread. But before this could awaken, the S2 message was received, processed by DRAMA and sent straight to the user. Only then, did the action thread wake up, process the trigger message and send message C2.

In traditional DRAMA, the trigger message would have been processed before DRAMA started looking for another message to process, and hence the order is always right.

This effect also impacts ERS messages – they can also get to the user out of sequence.

DRAMA does have techniques that would allow this to be avoided[20], but this does removes a significant DRAMA optimization. If this is an issue, please let the author know and the implementation will be adjusted.

[20] Future implementation note – `DitsInterested()` can be used, transferring the message to the thread. The MessageEventHandler object could then output them.

8 GIT Task Implementation

It is an AAO Software Standard that all Instrument tasks must obey the "Generic Instrument Task" (GIT) specification. Such a task provides a standard base action interface and a standard base set of parameters. The DRAMA Document "GIT_SPEC_9" defines this specification

The benefit of the GIT specification is that it makes it easy to write Control Task that can use standard code to Load, Initialise, Reset, Monitor and Shutdown Instrument tasks.

"GIT_SPEC_9" also describes a library that is used to assist in implement GIT tasks. Whilst written in C, it provided an O-O approach. You inherited GIT to get the basic interface. You might then inherit one or more other interfaces that may just add new features, or may override features in GIT. Finally, you add you own interfaces, which may again override GIT or other inherited features, or just simply add your own.

DRAMA2 re-implements the GIT interface using DRAMA2, via the "drama::git::Task" class. To implement a GIT task, sub-class this, adding your own actions and parameters.

The simplest possible GIT task implementation is shown in Example 7–1. Basically, you can just run the `drama::git::Task` constructor with `CreateRunDramaTask`. NOTE – unlike all previous examples, just including "`drama.hh`" is not sufficient, you must also include "`drama/gittask.hh`" to get access to the features. This is to avoid non-GIT drama tasks facing a higher compilation overhead.

Example 8–1. A Basic GIT Task.

```
10.  #include "drama.hh"
11.  #include "drama/gittask.hh"
12.
13.  const char *taskVer="1.0";
14.  const char *taskDate="21-03-2015";
15.  /**
16.   * Create the simplest  GIT task.
17.   */
18.  int main()
19.  {
20.      drama::CreateRunDramaTask<drama::git::Task>(
21.          "EXAMPLE8_1", /* Task name */
22.          "EXAMPLE",    /* Log sys name */
23.          taskVer,      /* Task version */
24.          taskDate,     /* Task date */
25.          "D2 Example 8.1 - A Basic GIT Task");
26.
27.
28.      return 0;
29.  }
```

And below shows the results of playing with this task using `ditscmd`. In particular note how the various arguments ended up in parameters. A control task can use these standard parameter values to work out what it is controlling.

```
>> ./exam8_1&
[3] 6529
>> ditscmd EXAMPLE8_1 SIMULATE_LEVEL FULL
DITSCMD_19a1:EXAMPLE8_1:Simulation has been set to FULL
```

```
>> ditscmd EXAMPLE8_1 INITIALISE
DITSCMD_19a2:EXAMPLE8_1:Opened log file
"/home/tjf/odc_temp/logs/EXAMPLE8_1-2015-03-10.00.log"
DITSCMD_19a2:EXAMPLE8_1:Initial logging levels set to ERRORS,INST,MSG
DITSCMD_19a2:EXAMPLE8_1:DRAMA Generic Instrument Task Initialised

>> ditscmd EXAMPLE8_1 RESET
DITSCMD_19a4:EXAMPLE8_1:Opened log file
"/home/tjf/odc_temp/logs/EXAMPLE8_1-2015-03-10.01.log"
DITSCMD_19a4:EXAMPLE8_1:Initial logging levels set to ERRORS,INST,MSG
DITSCMD_19a4:EXAMPLE8_1:DRAMA Generic Instrument Task Reset

>> ditscmd -g EXAMPLE8_1 _ALL_
DITSCMD_19cb:SdpStructure          Struct
DITSCMD_19cb:  LOG_LEVEL             Char   [16] "ERRORS,INST,MSG"
DITSCMD_19cb:  GITLOG_FILENAME       Char   [53]
"/home/tjf/odc_temp/logs/EXAMPLE8_1-2015-03-10.02.log"
DITSCMD_19cb:  SIMULATE_LEVEL        Char   [5] "FULL"
DITSCMD_19cb:  TIME_BASE             Float   1
DITSCMD_19cb:  ENQ_DEV_TYPE          Char   [4] "IDT"
DITSCMD_19cb:  ENQ_DEV_DESCR         Char   [34] "D2 Example 8.1 - A Basic
GIT Task"
DITSCMD_19cb:  ENQ_VER_NUM           Char   [4] "1.0"
DITSCMD_19cb:  ENQ_VER_DATE          Char   [11] "21-03-2015"
DITSCMD_19cb:  ENQ_DEV_NUMITEM       Int     0
DITSCMD_19cb:  INITIALISED           Int     1

DITSCMD_19cb:  POLL_PARAMETER        Float   2
>> ditscmd EXAMPLE8_1 EXIT
[3]  + done       ./exam8_1
```

8.1 Overriding GIT Action Implementations.

The example above is not very useful. In most cases, you will want to add your own actions and override GIT actions to implement the requirements of your task. For example, very few GIT tasks use the default `INITIALISE` action. They will implement their own to initialize the hardware or other internal state.

For both overriding GIT actions and adding your own, the process is the same and is exactly the process we have taken before to add actions. Just add the new actions from your task constructor.

Example 8–2, below, shows this being done. In this example, the new `INTIALISE` action is implemented using an action thread, line 1. The content of the `ActionThread()` method (line 10) shows the jobs done by the default GIT `INITIAILISE` action, opening the log file (line 13 – logging is covered in section 11) and setting the `INITIALISED` parameter to 1 (line 16). Whilst you should do those to jobs, the rest of the contents are up to the author and the job required. The `INITIALISED` parameter is used by control tasks that find a task already running when it tried to connect. If it is set true, the control task may then not bother Initialising/Resetting the task.

Example 8–2. Overriding GIT Action Implementations

```
1.   class InitialiseAction : public drama::thread::TAction {
2.   private:
3.       std::weak_ptr<drama::Task> _theTask;
4.   public:
5.       InitialiseAction(std::weak_ptr<drama::Task> theTask) :
```

```
6.          TAction(theTask), _theTask(theTask) {}
7.
8.      void  ActionThread(const drama::sds::Id & /*obeyArg */)
9.          override {
10.         // Access GitTask.
11.         auto myTask(GetTask()->TaskPtrAs<drama::git::Task>());
12.         // Open log file.
13.         myTask->Logger().Open(myTask->GetLogSysName());
14.         // Access par sys, set initiailsed.
15.         drama::ParSys parSys(_theTask);
16.         parSys.Put("INITIALISED", 1);
17.         MessageUser(
18.             "EXAMPLE 8.2 GIT Task Initialised from a thread.");
19.     }
20.     ~InitialiseAction() {}
21. };
22.
23. const char *taskVer="1.0";
24. const char *taskDate="22-03-2015";
25.
26. class MyGitTask : public drama::git::Task {
27. private:
28.     InitialiseAction InitActObj;
29. public:
30.     MyGitTask(const std::string &taskName) :
31.         drama::git::Task(
32.             taskName,
33.             "EXAMPLE",    /* Log sys name */
34.             taskVer,      /* Task version */
35.             taskDate,     /* Task date */
36.             "D2 Example 8.2 - GIT Task with own Initialise"),
37.         InitActObj(TaskPtr()) {
38.
39.         Add("INITIALISE", drama::MessageHandlerPtr(
40.                 &InitActObj, drama::nodel()));
41.     }
42. };
43. /**
44.   */
45. int main()
46. {
47.     drama::CreateRunDramaTask<MyGitTask>("EXAMPLE8_2");
48.     return 0;
49. }
```

You should refer to the DRAMA Document “GIT_SPEC_9” for more details on what GIT action should do.

8.2 Accessing Simulation.

The `drama::git::Task` class provides various methods which allow you to access the simulation of the task. An argument to the constructor allows you to set which simulation levels are acceptable. Use `git::Task::IsSimulating()` for a logical test of simulation and `GetSimulationLevel()` for the actual simulation level. `GetSimuationTimeBase()` can be used to get the simulation time base (intended to be a multiplier of the speed of the simulation compared to the real hardware). Note that the values are read from the parameters on demand.

8.3 GIT POLL Action

The GIT Specification requires a POLL action. The POLL action is defined as an action task is started by the Loader/Control task after it has finished initialization a GIT Task. The POLL action will run until Kicked or until it is otherwise stopped by the task itself (say as part of a RESET/EXIT action). The POLL action provides a way for a task to control things that are not done in direct response to actions (e.g. polling hardware status to update parameters).

The default GIT POLL Action will take over all orphaned transactions (except those which cannot be dealt with using the traditional DRAMA technique – see section 7.10). It will report messages to the user for all such events.

Whilst a sub-class could override the implementation of POLL itself, there are a number of callbacks for the most common cases which avoids the sub-class having to deal with the complexity of the general case. These can be overridden independently as required. See the table below:

Method	Description
`bool PollObeyOverride( MessageHandler *)`	Invoked on all `POLL` messages. If it returns `false`, then the default behavior will then occur. If it returns `true`, it is presumed the message has been handled and poll is rescheduled. This method allows a sub-class to override any particular messages it desires, whilst other messages have the default behavior. The default implementation returns `false`.
`bool PollKick( MessageHandler *)`	Invoked to handle kick messages. Returns `true` if the `POLL` action should complete, `false` if it should reschedule according to the `POLL_PARAMETER` value. Note – control tasks would expect a `POLL` action to complete when it receives a kick. Control tasks expect that a `POLL` action completes when kicked. You should change this behavior only based on arguments to the kick, since the standard control tasks don't set any argument. This method is normally used to implement some tidying up in response to the kick of `POLL`, before the action exits.
`void PollSignalEvent( MessageHandler *)`	This method is invoked if the `POLL` action receives a signal event, but only if `PollObeyOverride()` returned `false`. Signal events are one of the most common events for poll handlers to be interested in, so this handler allows a sub-class to override the

Method	Description
	behaviour without having to do all the work needed if it were to use `PollObeyOverride()`.
`void PollReschduleEvent( MessageHandler *)`	This method is invoked if the `POLL` action receives a reschedule event, but only if `PollObeyOverride()` returned `false`. Reschedule events are one of the most common events for poll handlers to be interested in, so this handler allows a sub-class to override the behaviour without having to do all the work needed if it were to use `PollObeyOverride()`.

It must be noted that implementing the above methods won't allow you to send messages from `POLL`, since the deault `POLL` action is not threaded. IF you need to send messages from `POLL` you will need to reimplement it from scratch.

8.4 GIT Path.

The job of the GIT specification is to make writing control tasks easier by providing a consistent interface. The `drama::git::Path` class helps implement control tasks - it takes advantage of the consistent interface provided by tasks obeying the GIT specification to provide an easy way to load and run GIT tasks.

`drama::git::Path` is a sub-class of `drama::Path`, so all the features of that area also available. The extra features from a control-task viewpoint are methods which wrap up the standard functionality of GIT tasks in a convent manner.

8.4.1 Initialise Method

The `drama::git::Path::Initialise()` method wraps up the process of loading a task, setting the simulation level and Initialising/Resetting the task.

Before executing this, the simulation level and reset type should be set. Defaults are no simulation and a reset type of "`FULL`". Details on how to do this are below (section 8.4.6).

The task is loaded if it is not already running. The simulation level is set and details of the task are fetched from the standard parameters.

If the task was loaded or if it was already running and the `INITIALISED` parameter has the value zero, then the task will be send an `INITIALISE` action.

If the task was already running and the `INITIALISED` parameter was set to a non-zero value, then the reset mode is considered. If the reset mode is "`RECOVER`", then a `RESET FULL` is done only if the task is in a failed state. Otherwise the requested `RESET` command is sent with the specified argument. Thus if the reset mode is "`RECOVER`", tasks will only be reset if they have failed.

`Initialise()` blocks until the operation is complete, so you need to execute this in a thread in the normal way.

8.4.2 Exit Method

The `drama::git::Path::Exit()` method will send the EXIT action to the task. It also does some internal accounting around doing this, and sets the object state to failed.

`Exit()` blocks until the operation is complete, so you need to execute this in a thread in the normal way.

8.4.3 Poll Method

The `drama::git::Path::Poll()` sends the `POLL` action to the task.

If POLL fails, it will be restarted, up to the number of attempts set by `Path::SetPollMaxAttempts()`, except that if it is clear the task has died or we are canceling polling, then it is not restarted.

If the POLL action is rejected due to it already being active, then it will be cancelled and restarted by this task. This feature allows our task to take over polling that was started somewhere else.

`Poll()` blocks until the POLL action is complete, so you need to execute this in a thread in the normal way.

8.4.4 PollCancel Method

The `drama::git::Path::PollCancel()` method will kick the POLL action in the task, and ensure that the `Poll()` method does not restart it.

`PollCancel()` blocks until the kick is complete, so you need to execute this in a thread in the normal way. The order in which `Poll()` and `PollCancel()` complete is undefined.

8.4.5 Report Method

The `drama::git::Path::Report()` method outputs to the user (via `MessageUser()`), some information on the status of the path.

8.4.6 Other Methods.

The `drama::git::Path::SetPollMaxAttempts()` method sets the number of times the class will attempt to restart polling if it fails. The default is 5.

The `drama::git::Path::SetSimulation()` method sets the simulation details to be sent to the task as part of `Initiailse()`. The default is "`NONE`".

The `drama::git::Path::SetResetMode()` method sets the type of reset operation if a task is reset (rather then initialized) when the `Initialise()` method is next invoked. The default is a "`FULL`" reset.

All the above methods in this section should be invoked before `Initialise()` method, since they impact its behavior.

The `drama::git::Path::SetFailed()` method should be invoked after a (subclass) operation fails in such a way that the task should be reset to recover.

8.4.7 Example Usage

The example source file `exam8_3.cpp` demonstrates the use of this class. It runs the programs from Example 8–1 and Example 8–2, implementing the actions `INITIALISE`, `POLL`, `POLLKICK`, `REPORT` and `EXIT`, which exercise each of the core features in `drama::git::Path`. The core idea it demonstrates is having some set of GIT tasks under control and operating on each of them in parallel using simple code.

We won't display `exam8_3.cpp` here, as it is a bit long with the significant points actually coming from the structure as a whole. Please refer to the source code itself.

8.4.8 Sub classing git::Path

A typical use of a `drama::git::Path` would be in a control task like the 2dF control task[21]. This task controls a large set of DRAMA tasks, most of which are GIT tasks.

Such a control task sets up a dynamic list/vector etc. of tasks to be controlled. If these are all GIT tasks, then it could use `git::Path` to consistently and, in parallel, load, initialise/reset, poll and shutdown the tasks.

But many of these tasks have specific functionality the 2dF control task must operate, in addition to their GIT functionality. So for each, the control task provides a sub-class which accesses the specific functionality. This is standard C++ fair.

The more interesting case is where the tasks to be controlled actually don't obey the GIT specification. In the 2dF control task case, there are various "display" tasks that are not GIT tasks. But the control task still wants to control them in a consistent manner.

The approach in these cases is that the core functionality of `drama::git::Path` is provided by a bunch of simple method which can be overridden. For example, there is a `SendInitialise()` method which actually sends the INITIALISE action to the task being control. If the task being controlled does not support the standard GIT `INITIALISE` action, then a sub-class of Path should override `SendInitialise()` to do the appropriate thing.

The methods you might want to consider overriding in these cases are given in the table below.

Method	Description	Default Behavior
`DoGetPath()`	Loads a task and gets a path to it.	Invokes `drama::Path::GetPath()`.
`SendSimLevel()`	Sends the `SIMULATE_LEVEL` action to the task.	Invokes `drama::Path::Obey()` with action name `SIMULATE_LEVEL` and the `simLevel` and `timeBase` arguments.
`SendInitiailse()`	Initialises the task.	Invokes `drama::Path::Obey()` with

[21] The 2dF control task does not use drama::git::Path, it predates it by 20 years. But the technique described is used in the 2dF control task, implemented using the older DRAMA interfaces.

Method	Description	Default Behavior
		action name `INITIALISE`
`SendReset()`	Resets the task.	Invokes `drama::Path::Obey()` with action name `RESET` and the reset mode argument value.
`GetVerInfo()`	Fetches the values of various parameters.	Invokes `drama::Path::Get()` for the parameters `ENQ_VER_NUM`, `ENQ_VER_DATE`, `ENV_DEV_DESCR`, `ENQ_DEV_TYPE` and `INITIALISED`, filling out the relevant member variables with their values.
`SendExit()`	Exits the task.	Invokes `drama::Path::Obey()` with action name `EXIT`
`Poll()`	Runs polling in the task.	Rather complicated, as needs to restart polling in some cases and handle poll being cancelled. The underlying message is `drama::Path::Obey()` with action name `POLL`
`PollCancel()`	Cancels polling. Sets `_pollCancelling` variable to `true`.	Invokes `drama::Path::Kick()` with action name.
`Report()`	Reports on the status of the object.	Outputs details from the parameters (`GetVerInfo()`) before invoking `drama::Path::Report()`.

It would be unusual to override `Initialise()` in a sub-class – there is a lost of functionality to be re-implemented.

Note that currently various class variables are available to sub-classes. Their use is described in the documentation. The intention is to replace direct access to these by appropriate methods, but it is unclear at the moment what interfaces are required.

9 Bulk Data

DRAMA provides a “Bulk Data” technique that allows you to send extremely large amounts of data using DRAMA. The size is only limited by the virtual memory limitations of the machines involved. First I shall explain the underlying problem and basic approaches before detailing how you use these techniques in DRAMA2.

When using the normal DRAMA message sending approaches, DRAMA writes the message header directly into the receiving task's buffers. This is very efficient for messages without argument structures. But, any argument structure you supply via a SDS id is exported from SDS into the buffer. This means that you must first put the data into an SDS item and then have DRAMA copy it into the buffer. As long as you don’t need to keep your argument around after your action entry – receiving is efficient, you can access the SDS argument directly from the buffer (but this doesn’t work for actions implemented as a thread – there it is copied). For large amounts of Data, say image arrays, this is an inefficient approach.

There are other problems. You are required to define the size of the largest message to be sent when you set up the path between two tasks. Also it is possible you need to read data from something like a frame buffer that can't be part of the DRAMA message buffers. The DRAMA “Bulk Data” feature solves all of these problems and works well with DRAMA 2 threaded actions.

The feature allows you to send very large amounts of data by specifying shared memory segments containing the data. For local transfers, this means no data is actually transferred, other then a small notification message sent to the target task. For network transfers, only one area of memory is required on each machine. The actual limit is determined by machine virtual memory restrictions. Bulk Data techniques may place extra requirements on the receiving task, depending on the particular example.

9.1 Sending Bulk Data

DRAMA uses the bulk data features of the underlying IMP system and you are referred to the IMP manual for implementation details. You don't actually call any IMP routines directly.

Some DRAMA message operations can be replaced by "Send Bulk Data" equivalents. In each case, the basic procedure is the same. You first define an area of shared memory. You then call the appropriate message sending routine specifying most of the normal arguments for the non-bulk routine and the shared memory segment details. You get notifications about the progress in sending/reading the bulk data, allowing the sender to know when it release or reuse the memory segment.

The bulk data segment can contain anything you want, but if it contains an SDS structure, the receiving task need not know anything about it being bulk data – it is a transparent operation to the receiver of a bulk data message containing an SDS structure, unless it explicitly looks for bulk data arguments.

9.1.1 Creating a Bulk Data shared memory segment

The class `drama::BulkData` is used to create a shared memory segment. This type has three constructors – the default constructor, the main constructor and an additional constructor normally only used by DRAMA 2 itself. The default constructor creates an item only suitable as the target for move operations. The rest of this considers the main constructor which generates a shared memory segment.

Such a shared memory segment has a type, indicated by the "`Type`" argument, using the enum `drama::ShareType`.

If `Type` is set to `ShareType::Create`, then this class's main constructor creates a temporary shared memory section of the size specified in some suitable form for the system it is running on. The caller has no control over the details of the shared memory section.

In this case, the Name, Key and Address arguments are all ignored. A new shared memory section will be created and mapped into the tasks virtual address space.

Under VxWorks: The mapped section is just a section of memory, starting at a specified address. Type should be `ShareType::Global`. Name should be an empty string, and Key is ignored. If `Create` is `true`, `Address` is ignored, and a suitably sized area of memory is allocated. If `Create` is passed as `false`, then `Address` should contain the address of the memory section in question. Under `VxWorks`, this is what is generated by `ShareType::Default`.

UNIX: The mapped section can be created as System V shared memory, in which case `Type` should be `ShareType::SHMem`, or as a file accessed through `mmap()`, in which case Type should be ShareType::MMap. If Type is ShareType::SHMem, `Key` specifies the identifier for the shared memory and `Name` should be an empty string. If Type is `ShareType::MMap`, Name should be the full name of the file, and `Key` is ignored. `Address` is ignored in both cases. On UNIX, one of these types will be what is provided by `ShareType::Default`.

The `drama::BulkData::Data()` method provides access to the raw data. This is a template method, allowing you to specify the type of object, which must be a POD type. There are move assignment and move constructor methods.

The sub-class `drama::BulkDataSds` is provided to support SDS structures within Bulk Data. Otherwise very similar, the main difference is that the constructor allows the size of the segment to be determined by the size an SDS structure would be if exported, and then exports the SDS structure into the segment. The important point here is that when you create an SDS item, the memory is not generated until required – the SDS item is not "defined" until needed. So you can create an SDS structure that represents a very large area of memory, but if you have not written data to it, it uses little memory. You provide such a structure to the constructor as the "Template" of the SDS item. The `drama::BulkDataSds` constructor uses this template to determine the required size and then exports the SDS item into the bulk data memory segment. The `drama::BulkDataSds` class is also a sub-class of drama::sds::Id.

9.1.2 Sending a Bulk Data trigger message

Bulk Data trigger messages can be sent using the `drama::MessageHandler::SendBulkTrigger()` method. A version is provided which works with `drama::BulkData` items, another works with `drama::BulkDataSds` items.

The main difference between a bulk data trigger message and a normal trigger message (other then the bulk data argument) is that the operation is not completed immediately from the senders viewpoint. In a normal trigger, once the `SendTrigger()` method returns, you are free to delete or reuse the SDS argument and complete the action. For a bulk data message,

you are not free to delete or reuse the argument until notified, and your action must reschedule to wait for the notifications.

Two relevant notification messages are possible. You may get a `EntryCode::BulkTransferred` notification. This is used to tell your task the progress of the receiving task in processing your message. In some cases, it may allow your task to start updating some of the memory. Your task can use `drama::MessageHandler::GetEntry().GetBulkInfo()` to get details on the transfer progress. This notification is optional, it is only sent if the receiving task reports it progress in using the data to the sender.

The second notification is sent when the receiving task is finished with the bulk data memory segment. This is the `EntryCode::BulkDone` notification. After this has been received, you task is free to reuse or delete the memory segment.

The `NotifyBytes` argument, if non-zero, indicates that the initiating action should be notified every time that number of bytes is transferred. This is a hint only to the task receiving the message, and may be ignored. It is these notifications that trigger the `EntryCode::BulkTransferred` notifications.

Example 9–1 is an example of sending a bulk data trigger message. The Action TEST in this task will do this. The Action implementation starts from line #20. At lines 22 to 34, a template SDS structure is created. In this case, structure containing an item named "`BYTE_ARRAY`", which is a 1024 x 512 array of bytes. This is used to create the shared memory segment at line 37. We are dynamically allocating the object in the static variable so that we can keep it about when the method returns (you should probably used a `std::shared_ptr<>` rather then just a pointer).

We then access the data within the shared memory and fill it with some values. Lines 40 to 50. As an aside here, we are using the `drama::sds::DataPointer` class to provide access to the raw data in the SDS structure.

Then at line 54, we send the trigger message. We need to pass a transaction DRAMA ID argument address to this method. A transaction ID allows us to relate a message sent to replies received. In this case, we don't need to use it, but other examples may do so. The transaction ID association with the resultant notifications could be obtained using `GetEntry().EntryTransId()` and compared to what was returned here.

Finally the action provides a new reschedule handler (line 57) and reschedules to await the reply. Real code may reschedule with a timeout rather then just a sleep.

Example 9–1. Sending a bulk data trigger message

```
1.   static drama::BulkDataSds *bigDataSds  = nullptr;
2.
3.   // Handler for bulk data trigger reschedule messages.
4.   class MyRescheduleHandler : public drama::MessageHandler {
5.   public:
6.       MyRescheduleHandler() {}
7.       ~MyRescheduleHandler() {}
8.       drama::Request MessageReceived() override;
9.   };
10.
11.  // Action definition.
12.  class TestBulkAction : public drama::MessageHandler {
13.  public:
14.      TestBulkAction() {}
```

```
15.      ~TestBulkAction() {}
16.  private:
17.      MyRescheduleHandler _reschedHand;
18.
19.
20.      drama::Request MessageReceived() override {
21.
22.          // Toplevel of template SDS items.
23.          drama::sds::Id myTemplate =
24.              drama::sds::Id::CreateArgStruct();
25.
26.          // Dimensions of template SDS data array.
27.          std::vector<unsigned long> dims(2);
28.          dims[0] = 1024;
29.          dims[1] = 512;
30.
31.          // Create array of bytes.
32.          drama::sds::Id childArray =
33.              myTemplate.CreateChildArray("BYTE_ARRAY",
34.                                          SDS_BYTE, dims);
35.
36.          // Create bulk data item.
37.          bigDataSds = new drama::BulkDataSds(GetTask(),
38.                                              myTemplate);
39.
40.          // Find the child array in the bulk data item .
41.          childArray = bigDataSds->Find("BYTE_ARRAY");
42.          // Fill the child data array with something
43.          drama::sds::DataPointer<char[]> dataArray(childArray);
44.          char val = 0;
45.          for (auto &item : dataArray)
46.          {
47.              val = val % 128;
48.              item = val;
49.              val ++;
50.          }
51.
52.          // Send trigger
53.          DitsTransIdType transid = nullptr;
54.          SendBulkTrigger(bigDataSds, &transid);
55.
56.          // Reschedule.
57.          PutObeyHandler(
58.              drama::MessageHandlerPtr(&_reschedHand,
59.                                       drama::nodel()));
60.          return  drama::RequestCode::Sleep;
61.
62.      }
63.  };
64.
65.  // Handle reschedule messages for bulk data trigger transfer.
66.  drama::Request MyRescheduleHandler:: MessageReceived()
67.  {
68.      switch (GetEntry().Reason())
69.      {
70.      case drama::EntryCode::BulkDone:
71.      {
72.          // Bulk data transfer finished.
73.          MessageUser("MyRescheduleHandler:: BulkDone");
74.          delete bigDataSds;
75.          bigDataSds = nullptr;
```

```
76.           return  drama::RequestCode::End;
77.       }
78.       case drama::EntryCode::BulkTransferred:
79.       {
80.           // Bulk data Transfer progress message.
81.           DitsBulkInfoType BulkInfo = GetEntry().GetBulkInfo();
82.
83.           double percent_done =
84.               (((double)(BulkInfo.TransferredBytes)/
85.                 (double)(BulkInfo.TotalBytes)) * 100.0 );
86.
87.           MessageUser("BulkTransferred:" +
88.                       std::to_string(percent_done) + "%");
89.           return  drama::RequestCode::Sleep;
90.       }
91.       case drama::EntryCode::Rejected:
92.       {
93.           // Bulk data transfer rejected.
94.           delete bigDataSds;
95.           bigDataSds = nullptr;
96.           DramaTHROW(GetEntry().Status(),
97.                      "Bulk data transfer rejected");
98.       }
99.       default:
100.      {
101.          MessageUser(
102.              "MyRescheduleHandler:: UNEXPECTED Message");
103.          break;
104.      }
105.      } // switch
106.      return  drama::RequestCode::Sleep;
107. }
```

The reschedule handler, from line 66, is basically a switch on the notification reason (line 68). When an entry with a reason of `drama::EntryCode::BulkTransferred` is received, it just reports the transfer progress (lines 79 to 89) and reschedules. `If drama::EntryCode::BulkDone` is received, it is free to release the bulk data shared memory segment (line 74). It is also possible the transfer to be rejected – line 91.

You can test this example using `ditscmd`. For example

```
>> ./exam9_1&
[1] 21664
>> ditscmd EXAMPLE9_1 TEST
DITSCMD_54b1:Trigger Message from Action  "TEST", Task "EXAMPLE9_1". value
= 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 .
DITSCMD_54b1:EXAMPLE9_1:MyRescheduleHandler:: BulkDone
>> ditscmd EXAMPLE9_1 EXIT
[1]  + done       ./exam9_1
```

In this case, "`ditscmd`" doesn't actually know it received a bulk data message. It just sees an SDS argument to a trigger message. Later code below shows how to deal with receiving bulk data when the default handling is not sufficient.

[Consider version of SendBulkTrigger which wraps up the wait, particularly when done from thread]

9.1.3 Sending a Bulk Data Obey/Kick message

You can send Obey or Kick messages including bulk data arguments. `drama::Path::ObeyBulk()` and `drama::Path::KickBulk()` are used. The argument is also created using `drama::BulkData` or `drama::BullkDataSds`.

This is actually simpler the trigger message case, as the blocking nature of the Obey/Kick methods allows everything to be wrapped up nicely. Example 9–2 shows how it is done. This example uses a convenience class for creating the bulk data – from lines 1 to 35, which makes the `ActionThread()` method very simple. See lines 48 to 54. Because everything is done when the `ObeyBulk()` method returns, the bulk data can be released immediately (by its destructor)

Example 9–2. Obey message with bulk data.

```
1.  // Convenience class for constructing a BulkDataSds item.
2.  class MyBulk : public drama::BulkDataSds {
3.  public:
4.      MyBulk(drama::Task *theTask) {
5.           // Create a template SDS item.
6.          drama::sds::Id myTemplate =
7.              drama::sds::Id::CreateArgStruct();
8.
9.          std::vector<unsigned long> dims(2);
10.         dims[0] = 1024;
11.         dims[1] = 512;
12.
13.         drama::sds::Id childArray =
14.             myTemplate.CreateChildArray(
15.                "BYTE_ARRAY", SDS_BYTE, dims);
16.
17.         // Create the bulk data item and move to this object.
18.         this->drama::BulkDataSds::operator=
19.             (drama::BulkDataSds(theTask, myTemplate));
20.
21.         // Find the child item so we can fill it.
22.         childArray = this->Find("BYTE_ARRAY");
23.
24.         // Fill the data array with something
25.         drama::sds::DataPointer<char[]> dataArray(childArray);
26.         char val = 0;
27.         for (auto &item : dataArray)
28.         {
29.             val = val % 128;
30.             item = val;
31.             val ++;
32.         }
33.
34.     }
35. };
36.
37. // Action definition.
38. class RunAction : public drama::thread::TAction {
39.     drama::Task *_theTask;
40. public:
41.     RunAction(drama::Task *theTask) :
42.         TAction(theTask), _theTask(theTask) {}
43.     ~RunAction() {}
44. private:
45.
```

```
46.      void ActionThread(const drama::sds::Id &) override {
47.          // Create a bulk data SDS shared memory segment.
48.          MyBulk bigDataSds(_theTask);
49.          // Find the path to the  program, loading it if needed.
50.          drama::Path server(
51.              _theTask, "EXAMPLE9_3", "", "./exam9_3");
52.          server.GetPath(this);
53.          // Send the SDS structure using bulk data.
54.          server.ObeyBulk(this, "TEST1", &bigDataSds);
55.
56.      }
57.  };
```

There are versions of `drama::Path::ObeyBulk()` and `drama::Path::KickBulk()` which take a `drama::BulkData` item for the argument rather then `drama::BulkDataSds` as used in example. There are also waiting versions of these. There are also "WaitUntil" versions of these methods; through you have to think hard about the bulk data item destructors when using these.

Testing of Example 9–2 will need Example 9–3 below.

9.2 Receiving Bulk Data

As mentioned above, if a message has Bulk Data argument that contains an SDS item, then the receiving task need do nothing – things will work as normal (but be more efficient).

But – the receiver can take more control over the memory segment.

9.2.1 Non-threaded actions

Actually – it is only non-threaded actions (at the moment), which can take more control when receiving bulk data messages. The object returned by the `drama::MessageReceived::GetEntry()` method provides the `ArgIsBulk()` method to allow an action reschedule event to determine if it has a bulk data argument. If this is true, then an object of type `drama::BulkDataArg` or is subclass `drama::BulkDataArgSds` can be constructed in the action. Both of these are subclasses of `drama::BulkData`, whist `BulkDataArgSds` is also a sub-class of `drama::sds`. The `BulkDataArg` class provides access to the raw memory segment that is the argument, whilst `BulkDataArgSds` presumes the memory segment contains an SDS structure and provides access to that.

The `GetNotifyBytes()` method of these classes allows you to fetch the requested notification interval, in bytes, that the sender has requested. The `Report()` method is the method used to actually report on usage (triggering an entry with code `drama::EntryCode::BulkTransferred` in the sender). This might be useful in say cases where you are processing an image and can allow the sender to reuse the area of the image you have already processed. The destructor releases the shared memory segment and it is at this point that the sender will get an entry with the code `drama::EntryCode::BulkDone`.

Example 9–3 is a relatively simple bit of code. The action TEST1 has checks to see if it has an argument (lines 12 and 14), and if yes, checks to see if it is bulk data (18). If yes, then it creates a `BulkDataArgSds` item (line 21) and lists it.

Example 9–3 Receiving Bulk Data

```
1.   class Test1 : public drama::MessageHandler {
2.   public:
3.       Test1() {}
4.       ~Test1() {}
5.   private:
6.
7.       /*
8.        * Invoked when the Obey message is received.  Just
9.        * ends immediately
10.       */
11.      drama::Request MessageReceived() {
12.          drama::sds::Id Arg = GetEntry().Argument();
13.          // Arg is only true for SDS argument.
14.          if (Arg)
15.          {
16.              MessageUser(
17.                  "TEST 1 action running and has argument");
18.              if (GetEntry().ArgIsBulk())
19.              {
20.                  MessageUser(" Argument is bulk data");
21.                  drama::BulkDataArgSds bArg(GetTask());
22.                  bArg.List(SdsListToUser());
23.              }
24.              else
25.              {
26.                  MessageUser(" Argument is not bulk data");
27.                  Arg.List(SdsListToUser());
28.              }
29.          }
30.          else
31.          {
32.              MessageUser("TEST 1 action running - no argument");
33.          }
34.
35.          MessageUser("TEST 1 complete");
36.          return  drama::RequestCode::End;
37.      }
38.  };
```

The same approach will work for Kick messages.

Below we see Example 9–2 and Example 9–3 being exercised with ditscmd.

```
>> ./exam9_3 &
[1] 20081
>> ./exam9_2 &
[3] 20082
>> ditscmd EXAMPLE9_3 TEST1
DITSCMD_4e74:EXAMPLE9_3:TEST 1 action running - no argument
DITSCMD_4e74:EXAMPLE9_3:TEST 1 complete

>> ditscmd EXAMPLE9_3 TEST1  "quick fox"
DITSCMD_4e75:EXAMPLE9_3:TEST 1 action running and has argument
DITSCMD_4e75:EXAMPLE9_3: Argument is not bulk data
DITSCMD_4e75:EXAMPLE9_3:ArgStructure         Struct
DITSCMD_4e75:EXAMPLE9_3:  Argument1            Char   [10] "quick fox"
DITSCMD_4e75:EXAMPLE9_3:TEST 1 complete

>> ditscmd EXAMPLE9_2 RUN
```

```
DITSCMD_4e76:EXAMPLE9_3:TEST 1 action running and has argument
DITSCMD_4e76:EXAMPLE9_3: Argument is bulk data
DITSCMD_4e76:EXAMPLE9_3:ArgStructure         Struct
DITSCMD_4e76:EXAMPLE9_3:  BYTE_ARRAY           Byte   [1024,512] { 0 1 2 3
4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ...}
DITSCMD_4e76:EXAMPLE9_3:TEST 1 complete

>> ditscmd EXAMPLE9_2 EXIT
>>
[3]  + done       ./exam9_2
>> ditscmd EXAMPLE9_3 EXIT
[1]  - done       ./exam9_3
```

9.2.2 Threaded Actions

There is (as yet) no special handling available for bulk data in threaded actions. But, in that case where a bulk argument contains an SDS item, that item is accessed from within the bulk data in an efficient manner – avoiding the copy which is required for non-bulk SDS arguments to threaded actions.

10 User Interfaces

In every example developed so far, the DRAMA code of interest has been executed within a DRAMA action, within a threaded or non-threaded action. The messages are processed in a "DRAMA Context", that of an Obey message or a Kick message. The examples so far have used the "`ditscmd`" program to send the messages. But how can you write something like "`ditscmd`"? This applies to any user interface that wants to send DRAMA messages.

DRAMA has the concept of a "User Interface" (UFACE) context to support sending messages from outside DRAMA to DRAMA tasks. The underlying C level concept is that before using one of the message sending routines, you invoke a particular routine (`DitsUfaceCtxEnable()`) which supplies a function which will be invoked to respond to reply messages, in the same way that an action's reschedule handler will be invoked to reply to messages sent by an action.

DRAMA 2 provides access to this feature. An important part of the DRAMA 2 design was to support modern GUIs (such as a JAVA based GUI), which use threads to dispatch events from the GUI. The GUI threads must be able to send DRAMA messages.

In DRAMA 2, the key is `the drama::thread::TUface` class. Like `drama::thread::TAction`, this class is a sub-class of `drama::thread::TMessHandler` and the address of an object of this type can be specified as the "action" handler object to each of the `drama::Path` methods which send messages. So almost all the message sending "Control Task" code we have seen in sections 7 to 9 can be rewritten using one of these objects instead of the `TAction` object.

A thread wanting to send a DRAMA message should first create an object of the drama::thread::TUface class. It can then send messages as required. The result is actually quite simple, and most of the complexity in our first example is about how the thread itself gets started and its results are handled

Example 10–1 below is a simple program that runs the program from Example 2–1 from a User Interface thread. The example code will get a path to the `EXAMPLE2_1` task, send it the `HELLO` action and then the `EXIT` action. It will then exit itself. In this, it is behaving in a similar way to "`ditscmd`".

A thread invoking the function `MyThread()` does the work. At line #8, it creates a `drama::thread::TUface` object. It can then get a path to the task (line #13_ and send the messages (lines 16 and 17).

To trigger the DRAMA Main loop to exit, it must invoke `drama::thread::SignalDramaToExit()`. Note – this is only necessary if you want the DRAMA Main loop to exit, typical of a simple command line program. For a proper GUI, this would not normally be done at this point, instead you would invoke `SignalDramaToExit()` as part of your GUI shutdown.

The reason for the try/catch block is that, in this example, if you don't invoke `SignalDramaToExit()` then the program will never exit. So even if an exception is thrown, you must ensure you invoke it. This type of thing is not an issue in action code, since DRAMA starts the action thread and tidies up correctly.

As for the rest of the implementation of this program, it is a touch more complicated. The `UfaceTask` constructor is required to start the thread that sends the messages (line #60).

But before doing that, it must block signals to avoid race conditions. You can do this constructing a `drama::thread::SignalBlocker` object, line #54. The signals will be restored when the object is destroyed. The future created by starting the `async()` thread is stored.

The real extra complexity in this example is in the shutdown. This is because we must ensure that any exceptions thrown by the thread are transferred to the main code, but we must also handle a failure of the thread to finish. The `WaitThreadExit()` method does this. The idea is that this is invoked after `drama::task::RunDrama()` returns, that is, the DRAMA task is shutting down. It first waits a number of seconds for the thread to complete. Since in most cases, the thread completion is what triggered the DRAMA task to shut down (call to `SignalDramaToExit()`), then this should happen quickly. It must then do a `get()` on the future, so that the thread is cleaned up correctly and any exception thrown by the thread can be propagated.

To ensure this is done after drama::task::RunDrama() exits, the UfaceTask class overrides RunDrama() (line 84), with its implementation calling the original before it invoked `WaitThreadExit()`.

Example 10–1. Basic User Interface Example

```
1.   void MyThread(drama::Task *task)
2.   {
3.       try
4.       {
5.           // We must create a ufaceHandler at this point.
6.           //  Only then can we send DRAMA messages.
7.           //  Also block all signals to the thread.
8.           drama::thread::TUface ufaceHandler(task);
9.
10.          // Find the parth to the EXAMPLE2_1 program
11.          drama::Path server(task, "EXAMPLE2_1",
12.                             "", "./exam2_1");
13.          server.GetPath(&ufaceHandler);
14.
15.          // Send messages.
16.          server.Obey(&ufaceHandler, "HELLO");
17.          server.Obey(&ufaceHandler, "EXIT");
18.
19.      }
20.      catch (...)
21.      {
22.          // Tell DRAMA To exit - causes task to exit
23.          drama::thread::SignalDramaToExit(task);
24.          throw;
25.      }
26.      // Tell DRAMA To exit - causes task to exit
27.      drama::thread::SignalDramaToExit(task);
28.  }
29.
30.  /* DRAMA UFACE example task.
31.   */
32.  class UfaceTask : public drama::Task {
33.
34.  private:
35.      // Future for the thread running the task.
36.      std::future<void> _ufaceThreadFuture;
37.  public:
38.
```

```
39.      /**
40.       * Constructor, from here we add actions
41.       *  and start the thread.
42.       */
43.      UfaceTask(const std::string &taskName) :
44.                drama::Task(taskName) {
45.
46.          Add("EXIT", drama::SimpleExitAction);
47.          /*
48.           * By constructing one of these, we block all
49.           * signals whilst the thread is being created,
50.           * to avoid race conditions. The thread will block
51.           * signals itself once it starts running. The
52.           * destructor restores the signal mask.
53.           */
54.          drama::thread::SignalBlocker threadSignalBlocker;
55.
56.          /*
57.           * Launch the thread.
58.           */
59.          _ufaceThreadFuture = std::async(std::launch::async,
60.                                          MyThread, this);
61.      }
62.      /*
63.       * Used to wait for the thread started in the
64.       *  constructor to exit.
65.       * Timeout in seconds supplied
66.       */
67.      void WaitThreadExit(unsigned seconds) {
68.          // Wait for thread to exit.
69.          if (_ufaceThreadFuture.wait_for(
70.                  std::chrono::seconds(seconds)) !=
71.              std::future_status::ready)
72.          {
73.              DramaTHROW(DRAMA2__THREAD_TIMEOUT,
74.                         "UFACE Thread did not complete");
75.          }
76.          // Get on future. Will throw if async thread did.
77.          _ufaceThreadFuture.get();
78.      }
79.
80.      ~UfaceTask() {
81.      }
82.      // Override RunDrama, so we can wait for
83.      //  the thread to exit.
84.      void RunDrama() override {
85.          drama::Task::RunDrama();
86.          WaitThreadExit(3);
87.      }
88.  };
89.
90.  /** Program main.
91.   *
92.   */
93.  int main(int /*argc*/, char * /*argv*/[])
94.  {
95.      drama::CreateRunDramaTask<UfaceTask>("EXAMPLE10_1");
96.      return 0;
97.  }
98.
```

It should be possible to implement any user interface using the techniques from Example 10–1. It is important to remember that MyThread() does not complete until the message has returned. If the return of the thread is important for GUI response, please consider the impact and if a child thread is required.

11 Logging

Logging is an important feature of DRAMA. Complex multitasking systems require logging to allow tracing of complex interoperability problems and task errors. In traditional DRAMA, this is provided by the `GitLogger` C++ class. This class provides an interface which supports different logging levels, allows defaults to be set via environment variables and the current logging level to be changed by sending an action message to a task. But the `GitLogger` class is not threads friendly.

The `drama::logging::Logger` class is the DRAMA 2 equivalent. It is very similar to the old `GitLogger` class, and produces a similar log file.

An object of this type is created automatically as part of the creation of a DRAMA 2 `drama::Task` object. User code must invoke the `Open()` method to actually open the log file. That can be done in an action or the main line code. From your task object, you access the logger using the `drama::Task::Logger()` method.

The system has the concept of "logging levels". For example, you may want some things logged all the time, but others only when debugging a certain feature. Internally, the "logging levels" are a mask. During the invocation of a `Log()` call, the current logging levels are "and-ed" with the levels in the `Log()` call, and the message is written to the log file only if the result is non-zero. This means your task can have a lot of potential logging calls, but most of it can be disabled most of the time, able to be turned on by an action on demand.

11.1 Simplified Usage

When your task is created, it creates a `drama::logging::Logger` object. The constructor of this will create an action named `LOG_LEVEL` and a parameter also named `LOG_LEVEL` (if it does not already exist), allowing a task to meet the GIT specification. Another parameter named `GITLOG_FILENAME` is also created.

You can access the logger using the `drama::Task::Logger()` method. During your task constructor or during an action (such as `INITIALISE/RESET`), you should invoke `<task>.Logger().Open(<<system_name>>)` where `<<system_name>>` is a "system" name used for finding environment variables. The `<<system_name>>` is normally related to the default task name, but need not be.

The `LOG_LEVEL` action can be used to change the logging levels. The argument is a comma-separated list of levels represented as a string. The specified levels are added to the levels to be logged. The logging of a level can be turned off by prefixing its string name with "`NO`". The `LOG_LEVEL` parameter has the current values.

Use the `<task>.Logger().Log()` or `.SLog()` methods or the to actually log messages to the file. The `.Log()` versions are more efficient but use C `printf()` style formatting for arguments and hence are not type safe. The `.SLog()` versions use a type safe approach implemented using string streams, but are less efficient.

11.2 Environment Variables

The initialization of `drama::Logger` is controlled by number of environment variables, the same set used by `GitLogger`. The table below describes these.

Environment Variable	Default Value if not set.	Description.

Environment Variable	Default Value if not set.	Description.
<system_name>_LOG_LEVEL	Defers to DRAMA_LOG_LEVEL below	Contains the initial logging levels for the task. A comma separated list of log level names.
DRAMA_LOG_LEVEL	Defers to the LOG_LEVEL parameter.	Contains the initial logging levels for the task. A comma separated list of log level names.
DRAMA_LOGDIR	Value supplied to the Open() call. If that is null, then the current directory.	Supplies the name of the directory where the log file is to be written.
GIT_LOGGER_COMPRESS	Value supplied to the Open() call.	If set, then the log file will be compressed
GIT_LOGGER_NOCOMPRESS	Value supplied to the Open() call.	If set, then the log file will not be compressed even if the Open() call indicates it should be.

11.3 Log file locations/naming/day rollover.

The log file is written in the directory specified as the second argument to drama::logging::Logger::Open(). If this is specified as an empty string, then the DRAMA_LOGDIR environment variable specifies the directory. If this is not defined, then the current working directory is used. It is AAO practice to use the DRAMA_LOGDIR environment variable approach. It is appropriate that this directory be in a local disk, not a network disk, to ensure good performance.

The log file name has for format

{taskname}-{date}.<ver>.log.[.gz]

Where

{taskname} is the DRAMA name of the task.

{data} is the current *UT* date in the format "YYYY-MM-DD". This format allows an appropriate default order. The log file system uses UT time/dates throughout to ensure a consistent log file over events such as changes in daylight savings (summer time).

<ver> is the log file version number. Each time a task calls Open(), the version number will be incremented. Starts at "00". A new version will be started each time the file reaches about 1 GB in size.

`[.gz]` will be appended if the log file is compressed on write.

For example, if `DRAMA_LOGDIR= /home/tjf/odc_temp/logs/`, then:"

```
/home/tjf/odc_temp/logs//GITTEST-2015-03-16.06.log
```

is the sixth log file opened by the task GITTEST on the UT date 16th of March 2015.

11.3.1 Day Rollover

The name of the log file is based on the current UT date. The first time a message is written to the log file after a change in the UT date, the current log file is closed an a new log file is opened.

11.4 Actions/Parameters

11.4.1 LOG_LEVEL Action

The LOG_LEVEL action is used to set the logging levels. The argument to the action is a comma-separated list of levels represented as a string. The specified levels are added to the levels to be logged. The logging of a level can be turned off by prefixing its string name with "`NO`". See section 11.5, below, for the list of levels and their string names.

Level names can be abbreviated to the minimum significant numbers of characters and case is not significant.

11.4.2 Parameters

Two parameters are added to the task by `drama::logging::Logger`. The `LOG_LEVEL` parameter contains the current logging levels.

The `GITLOG_FILENAME` parameter will contain the name of the log file.

11.4.3 Related control messages

DRAMA Control messages, see section 7.13, provide some access to the logger. In particular, you can write a message directly to the log file (LOGNOTE) and get some details from it (LOGINFO). You can also flush the log file on deman(LOGFLUSH).

11.5 Logging levels

The table below lists each of the levels and the macro used to represent them. Also listed is the name of the equivalent macro in calls to `DitsLogMsg()`, where these is one.

Level	Macro	`DitsLogMsg()` equivalent	Description
STARTUP	`D2_LOG_STARTUP`	`DITS_LOG_STARTUP`	Log startup messages.
ERRORS	`D2_LOG_ERRORS`		Log errors - messages output by ERS and details of actions completing with bad status.
ACTENT	`D2_LOG_ACTENT`		Log Action and UFACE entry. Used by

Level	Macro	DitsLogMsg() equivalent	Description
			DRAMA logging
ACTEXIT	D2_LOG_ACTEXIT		Log Action and UFACE return. Used by DRAMA logging.
INST	D2_LOG_INST	DITS_LOG_INST	Log instrument specific functions.
COMMS	D2_LOG_COMMS		Log DRAMA communications (sending of messages etc).
ARGS	D2_LOG_ARGS		If logging action entry and/or action exit/comms, also log a string giving the details of any argument. For comms logs, this is only arguments to a send (obey etc).
DEBUG	D2_LOG_DEUG		Log debugging messages
DRAMA2	D2_LOG_DRAMA2		Log DRAMA 2 internals.
USER1	D2_LOG_USER1	DITS_LOG_USER1	User defined level 1. Equivalent to
USER2	D2_LOG_USER2	DITS_LOG_USER2	User defined level 2. Equivalent to
USER3	D2_LOG_USER3		User defined level 3.
USER4	D2_LOG_USER3		User defined level 4. For DRAMA 2 tasks, when this is specified in the DRAMA_LOG_LEVEL environment variable value, this is equivalent to the DRAMA2 level.
ALL	D2_LOG_ALL		All levels are defined.

It should noted that when specifying the DRAMA_LOG_LEVEL environment variable value in a system which might have older DRAMA tasks, specifying the DRAMA2 value will cause the older tasks to fail. Instead specify the "USER4" value, which is translated to the DRAMA2 value (but only if found in the DRAMA_LOG_LEVEL environment variable).

11.6 Opening the log file.

The `drama::logging::Logger::Open()` method takes three arguments, two of which has defaults. The first argument is the system name, used to determine the actual name of the `<system_name>_LOG_LEVEL` environment variable. This is normally set to a value related to the default task name, but where a given task may run under different names, you will probably want this value to be the same for all cases.

The second argument is the directory into which to put the log file. If an empty string is supplied, it will use the value of the `DRAMA_LOGDIR` environment variable, and if that has no value, the current default directory. Defaults to an empty string.

The third argument is a logical. If set true, then the log file is compressed. Defaults to false. If you compress your file, it may take longer to write (depending on CPU vs disk space) and may be harder to search.

11.7 Logging Messages

Most logging is likely to be done with the `drama::logging::Logger::Log()` or `drama::logging::Logger::SLog()` methods. The former takes four or more arguments. The first is the mask of log levels. This is used to indicate when this message should be written to the log file. For example, if the value is

```
D2LOG_DEBUG|D2LOG_USER1
```

Then it is written if the current log level is DEBUG or USER1.

The second argument is a logical. Set true to indicate the forth argument is a plain string (`const char *`), rather then something with C `printf()` style formatting. If the function author knows the string does not need formatting, it is much quicker to tell the method.

The third argument is a message prefix. This is a string of at most 20 characters, which is written to a particular field in the log file with the message. It is normally used to indicate the routine/method etc from which the log message was written.

The fourth argument is a C `printf()` style format string. Arguments 5 onwards are the arguments to the `printf()` format string.

An example call to the Log() is shown below. Note, user code would not normally use the D2LOG_DRAMA2 level. The INST, DEBUG and USER<n> levels are more appropriate for user code.

```
   _theTask().Logger().Log(D2LOG_DRAMA2,  false,
                           "d2::Path::GtPthImmd",
                           "Object %p:Get path to %s",
                            (void *)this, _taskName.c_str());
```

11.7.1 SLog()

C `printf()` style format strings don't fit very well with C++. They do provide a quick way to format simple C types and strings, but don't work with classes. The `logger.SLog()` methods are one approach to avoiding them. These methods use a neat, but probably run-time inefficent (I haven't tested the performance) approach that is type-safe and can format any argument which provides the standard "<<" operator to a `std::ostream` object. The format string in this case just has a single "`%`" character for each argument. The above example would become:

```
   _theTask().Logger().SLog(D2LOG_DRAMA2,"d2::Path::GtPthImmd",
```

```
                                "Object %:Get path to %",
                                 (void *)this, _taskName);
```

Note that you don't need things like "%d" (for an integer), just the "%" is needed as the "<<" operator knows how to format the argument. (In fact, in this case, the "d" will errantly appear in the output.)

Note that the second logical argument of `logger.Log()` has disappeared – it is not needed. One problem of of `logger.SLog()` is that if you get the number of arguments wrong, an error message occurs at run-time rather then compile time (For the `printf()` approach, many modern compilers can work out if you go that wrong). A second problem with `SLog()` is that you have somewhat limited control over the formatting – see §11.7.1.1 below.

There is another overload of `logger::SLog()` that adds an extra argument before the logging level. This extra argument is an output stream, of type `std::ostream &`. If this version is used, the log message is also written to that stream. The author has found this useful when developing an application where many log messages may be wanted on stderr during development, but are only wanted in the log file during production. To acheive this, the author will, when finished development, just do a global replace of the string ".`SLog(std::cerr`", with ".`SLog(`".

11.7.1.1 Limited formatting with SLog() via Manipulators

You can achive some formatting control when using `logger::SLog()` by using stream manipulators as arguments. To make this work, you add an extra "%" character at the point in the string where you want to change the formatting. You then supply the manipulator as the corrresponding argument. But please note that putting two "%" characters together does not work as you might think – it produces one "%" in the output, you have to work around this limitation. See §11.10.1.1 for some more details and a related example.

If this is not sufficient, you will have to use either the `Log()` method or the "Log Stream Buffer" approach, below.

11.7.2 Log Stream Buffers

Another way around the C printf issues of logger::Log() is to use the `drama::logger::LogStreamBuf` class. You can use this to create a `std::ostream` object, and write to this using standard stream methods and get all for formatting features of a stream.

Use the logger to create a `LogStreamBuf` object, which you then use to create the `std::ostream` object – see the example below. Messages are not actually written to the log file until either a `std::endl` is written, or `LogStreamBuf::flush()` is invoked, or the destructor is run. The time tag associated with the log entry is when the line is written to the file (e.g. `std::endl` is output or a flush happens), not when the first part of the message was written.

The code below shows how to use this. The second and thread arguments to the constructor are the logging level and prefix, as per the first and third arguments to `logger::Log()`.

```
drama::logging::Logger &logger =
    std::shared_ptr<drama::Task>(_theTask)->Logger();
```

```
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;
```

11.8 Interaction with DITS debugging.

The DRAMA DITS C library provides the function `DitsLogMsg()` for user logging. This call allows libraries to be written which can log to whatever logging system the task is using. If that task does not have a logger, the call is a null operation. The table above lists the macros which can be used with `DitsLogMsg()`.

Additionally, the DITS library has internal debugging facilities. This is controlled by the `DitsSetDebug()` routine, or the `DITS_DEBUG` environment variable (or control messages). See `DitsSetDebug()` for details of levels etc. If internal DITS debugging is enabled, and the task has a logger, the messages are written to the log file. If the task has no logger, then they are written to `stderr`.

The DRAMA2 logger will actually write these to `stderr` if the log file is not open, so that a task that does not open a log file behaves as per a task without a logger.

11.9 Log file content.

The log file contains 6 column separated columns, but it must be noted that the Time specification includes its own two colons.

Column	Size (Characters)	Description
Time of Message	19	The time of the message. Format “DD-MMM hh:mm:ss.sss”
Action Name	21	Name of action. Most also have the string `"—SYNC-EVENT—"` indicating a sync to disk event, `"—UFACE-CONTEXT—"` for a message from UFACE Context, `"—DITS-FIXED-PART—"` for a message from the DITS (DRAMA) code. `"—UNKNOWN-THREAD—"` for a message from a thread is not known to the logging system. The `RegisterThread()` method can be used to associated a user created thread with an action it is part of.

Column	Size (Characters)	Description
Thread ID	16	The thread ID
Thread Func	16	The thread function name (actually just the function name string passed to `logger.RegisterThread()`.)
Sequence	10	The action sequence number. Will be -1 for things such as sync events. Will be "T" for a message from a thread, other then the thread executing `drama::Task::RunDrama()`.
Message Prefix	20	User supplied message prefix
Message	Unlimited	The actual log message.

An example of typical log entries (with excess spaces removed and the thread thread function field removed as otherwise it gets to long) is below.

```
---------- task LOGTEST ----------------------
Time of Message    :Action Name      :Thread ID :Sequence Message Prefix  :Message
02-Apr 02:17:08:786:--UFACE-CONTEXT- :4158920400: 0:d2::Logger    :Opened log file
02-Apr 02:17:08:786:--UFACE-CONTEXT- :4158920400: 0:d2::Logger    :Logging levels set to
default values
02-Apr 02:17:08:786:--UFACE-CONTEXT- :4158920400: 0:d2::Logger    :*** Logging levels set, old
= c0022, new = c0032
02-Apr 02:17:08:786:--UFACE-CONTEXT- :4158920400: 0:d2::Logger    :Initial logging levels set
to ERRORS,INST,MSG
02-Apr 02:17:08:786:--UFACE-CONTEXT- :4158920400: 0:d2::Logger    :Task Version is r1_32, date
01-Apr-2015
02-Apr 02:17:08:786:--UFACE-CONTEXT- :4158920400: 0:d2::Task::RunDrama  :Invoked
02-Apr 02:17:20:229:--SYNC-EVENT-    :4158920400:-1:              :LogFlush() method (Probably
DitsMsgReceive())
02-Apr 02:17:48:155:TEST1            :4155722640: T:ERS           :Task TESTTASK not known
locally and no node supplied to find it on
02-Apr 02:17:48:155:TEST1            :4155722640: T:ERS           :Annulling 1 messages at
context 2
02-Apr 02:17:48:168:TEST1            :4158920400: 1:ERS           :DRAMA Exception thrown and
reported via ERS
```

The task version number and dates are retrieved from the GIT `ENQ_VER_NUM` and `ENQ_VER_DATE` parameters, if they exist. There are a few lines, such as the first two, that don't follow the format.

11.10 Type and thread safe printf style output to stdout/stderr.

A slight diversion from logging to simple output to `stdout` or `stderr`, as is often done whilst debugging. First note that any output from you task that is intended for the user should normally be done via `MessageUser()` or the exception handling or ERS systems. These methods ensure the message gets back to the user. There is also the `drama::MessageUserStream` class, which can be used to contruct a `std::ostream` sub-class that can be used to output such messages.

But whilst debugging a task, output to stdout or stderr may be required. The normal choices are the C `printf()` style functions – simple but unable to handle C++ objects – or C++ output to streams std::cerr/std:cerr. The later are safer then the `printf()` style functions, but at times the result is very complicated.

As seen in section 11.7.1 above, DRAMA2 has some ability to output safely in a `printf()` style way. The features are used to implement `drama::logging::SLog()` and for

internal debugging are also available for general use. These fuctions use a neat, but probably run-time inefficent (I haven't tested the performance) approach that is type-safe and can format any argument which provides the standard "<<" operator to a `std::ostream` object. The format string in this case just has a single "`%`" character for each argument. The first of the functions is `drama::SafePrintf()`, a simple print to a stream. For example:

```
drama::SafePrintf(std::cerr "Object %:Get path to %\n",
                  (void *)this, _taskName);
```

Which many would find easier to read then the standard streams approach of:

```
std::cerr << "Object " << (void *)this << ":Get path to "
          << _taskName << std::endl;
```

Note that you don't need things like "`%d`" (for an integer), just the "`%`" is needed as the "<<" operator knows how to format the argument. (In fact, in this case, the "`d`" will errantly appear in the output). Also that one problem of of these functions is that if you get the number of arguments wrong, an error message occurs at run-time rather then compile time (For the printf() approach, many modern compilers can work out if you go that wrong). A second problem is that you have limited control over the formatting (see §11.10.1.1, below). Finally, you need to specify a "`\n`" to flush the line.

In addition to `drama::SafePrintf()`, there is also `drama::TSafePrintf()`. This is probably to be preferred in any task with a lot of threads active. The `drama::TSafePrintf()` function ensures that for each call, all the output from that call is output in one output operation. That is, thread switching will not (normally) cause the a line be broken. There is still some small scope for such issues, but they have not been seen in practice.

Finally – it should be noted that `drama::SafePrintf()` can be used to write to a string stream (`std::stringstream`). That might be usefull in cases where you need to format into a string safely.

11.10.1.1 Limited formatting control with SafePrintf() via Manipulators

You can achive some formatting control when using `drama::SafePrintf()` through it has some limitations. First, you can apply stream manipulators to the stream before calling `SafePrintf()`. The problem with that is that you then can't change formatting control during the output of the line (e.g. you can't format different floating point numbers in different formats).

The other approach is to apply the manipulators as arguments. To make this work, you add an extra "%" character at the point in the string where you want to change the formatting. You then supply the manipulator as an argument at the corresponding point. See the following example

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

The above example will format the floating point number as “12.33”. The main issue with this approach is that you can’t put two “%” characters immediately next to each other (it results in one such character being output). The above example puts the formatting “%” character immediately after the comma, allowing a space before the floating point number is place. This works well in this case, but may not in others.

12 Internals

Currently a ramble through some features. May be extracted to its own document.

12.1 Action Threads

An action which wants to be implemented using a thread, must be a sub-class of `drama::thread::TAction`. This is itself a sub-class of `drama::thread::TMessHandler` and `drama::MessageHandler`.

The sub-classing of `drama::MessageHandler` allows such objects to be specified as an action handler object, with the `MessasgeReceived()` method being implemented to handle action obey messages.

The user sub-class of `TAction` must implement the `ActionThread()` method.

The result of the call to `MessageReceived()` is that the `ActionThread()` method is invoked within a new thread.

`MessageReceived()` will first installed new Obey and Kick reschedule handler objects to handle future events (`_obeyRescheduleObj` and `_kickMessageObj`). The former will result in `TAction::ObeyReschedule()` being invoked, the later `TAction::KickMessage()`.

The thread is then started and will run the "`thread::TAction::RunActionThread() method`". This will run the user's `ActionThread()` method. It then invokes the `ActionThreadComplete()` which will signal DRAMA. This is done even on exceptions. When an exception is thrown by `ActionThread()`, it is re-thrown after the call to `ActionThreadComplete()`.

12.1.1 ActionThreadComplete()

This must look for any threads which are waiting for drama messages and let them know they will never arrive – done by calling `TAction::SignalWaitingTheads()`.

Is then signals DRAMA using `TAction::SignalDrama()`, which causes a message to be put on the `_signalQueue` before a DRAMA2 signal is sent to the task.

12.1.2 ObeyReschedule()

This is invoked on any reschedule of the action (other then a kick message). So it is responding to messages from subsidiary tasks, GetPath messages. Of particular interest is the DRAMA 2 signals processed by `TAction::ProcessDrama2Signal()`.

12.1.3 ProcessDrama2Signal()

If invoked with a signal of Complete, will join the thread by invoking get() on the future associated with it. This will cause any exception to passed through.

Otherwise just reset the timeout.

12.2 Sending Messages from threads

Thread must either be implemented via a `thread::TAction` object (Threaded action) must create a `thread::UFace` object (UFACE thread). Both of these inherit from `thread::TMessHandler`, which provides an interface to the common messaging facilities.

12.2.1 Type message – Obey()

The first argument is the `thread::TMessHandler` object to use. This is passed to the "`Send`" method.

Fifth argument is the event processor – class `thread::MessageEventHandler`. This is a sub-class of thread::TransEvtProcessor, which is used to process message transaction events. `thread::MessageEventHandler` implements `Process()` and calls its own methods for each type of response message.

12.2.1.1 Send()

After some checks on the path etc, changes context to the action/uface context.

Sends the message (`DitsInitiateMessage()`).

Invokes `NewTransaction()` on the event handler.

Invokes `SetupWaitEvent()` on `messHandler` (`thread::MessageEventHandler` Object).

12.2.1.1.1 SetupWaitEvent()

Both the `TAction` and `UFace` versions of this invoke `thread::SetupWaitEvent()`. They provide the address of their own versions of the `_waitEventMap` object. The `UFace` version ensures the TID is not null.

The `_waitEventMap` object is a mapping index by thread ID, giving us event details. There can only be one event (transaction) associated with a given thread. But, a given `TAction`/`UFace` object may have multiple threads running and hence multiple transactions outstanding.

If we don't already have a map entry for this thread, create one. If we do have one, we expect it to be idle (not waiting) and update the details. If it is still waiting – programming error.

The `_waitEventMap` items are `WaitEventDetails` structures. This includes details of the transaction and a condition variable used to block threads on. The condition variable is stored as a `std::shared_ptr`, so we are only moving pointers about and the last one will clean it up.

Each `WaitEventDetails` item has a queue of events for the thread.

12.2.1.2 WaitForTransaction()

This will then be invoked on the `thread::MessageEventHandler` Object).

Basically we wait for the condition associated with the thread (via the WaitEventDetails) to be set. There is a queue of items here. When woken up, we grab the next item and process it using the event processor. We continue until the event processor indicates we are done.

An important point here is the management of the DRAMA lock. It is taken throughout, except whilst waiting for the condition.

12.2.2 ProcessSubsidiaryMessage

This is invoked by the TAction code when a reply to a message to a subsidiary task is received. Invoked by UFace code under similar conditions. The each have their own versions, but are very similar (unclear if they need to be different).

It finds entries in _waitEventMap waiting on this transaction and notifies them. This will cause the condition wait to return.

Before notifying the condition, it will push details of the transaction onto the queue.

12.3 Dealing with shutdown.

Threads complicate shutdowns. If a thread throws an exception that is not caught – the whole program will be terminated. The std::future class can be used to return the vale of a function executed as a thread and to transfer exceptions to another thread.

Action threads are implemented via std::async() functions, and therefore are associated with a future. UFACE threads can be launched by any means, but all examples provided use `std::async()`.

The `drama::thread::TAction` and `drama::thread::TUface` classes both implement the `drama::thread::RunDramaExitNotifier` interface.

Each time a UFACE or Action thread is started, it invokes `drama::Task::NotifyOnRunDramaExit()`, passing the address of itself to that method. The drama::Task class maintains a set of such objects. After the DRAMA main loop has exited, it invokes the `RunDramaExitNotifier::RunDramaHasExited()` method, allowing notifications to threads that they should exit immediately.

In both the default implementations, any thread waiting for a DRAMA message (is blocked `in drama::thread::WaitForTransaction()`) is notified by sending it an event with the entry code `drama::EntryCode::DramaRunLoopExit` and the status `DRAMA2__RUN_LOOP_EXIT`.

This will cause `drama::thread::WaitForTransaction()` to invoke its event processor's `drama::thread::TransEvtProcessor::Process()` method. For messages other then getting the path, this will results in `drama::MessageEventHandler::ThreadWaitAbort()` being invoked, which will throw an exception.

After the DRAMA main loop has exited and invoked the `RunDramaHasExited()` method on each `RunDramaExitNotifier`, it will invoke the `RunDramaExitNotifier::JoinThreads()` method on each `RunDramaExitNotifier`, allowing any threads to be joined before the program exits (if you don't join a non-detached thread, the program may not shutdown).

12.3.1 Normal Action Thread Shutdown

An action thread is created by running the `drama::thread::MyThread()` method, a friend of the `drama::thread::TAction` class (implemented in threadaction.cpp).

When this completes, it will invoke the method `TAction::ActionThreadComplete()` even when it completes with an exception). That method will signal any waiting subsidiary threads so that they may complete. It will also signal the DRAMA may loop to wake up by sending a DRAMA2 Signal with a message type of ThreadSignalType::Complete.

The `TAction::ProcessDrama2Signal()` method will be invoked in the main thread. It will do a `get()` on the `std::future` associated with the action thread. If the thread completed with an exception, it will now be delivered to the main thread that converts it to a DRAMA action error report and status.

12.3.2 Normal UFACE Thread Shutdown

UFACE threads are more complicated when Action Threads, as DRAMA cannot presume anything about how they have started. To become a UFACE thread, any thread just creates an object of the `drama::thread::TUface` class. can then use this in place of a `drama::thread::TAction()` object in all of the drama::Path methods that require such an argument

DRAMA has no control over which type of threads are involved or even if they remain as DRAMA UFACE threads during their entire run direction. It.

The TUface class destructor will ensure any outstanding message events are orphaned. It also ensures the object is no longer notified on DRAMA task exit.

In simple examples, where the thread running `drama::Task::RunDrama()` is now expected to exit, the thread should invoke `drama::thread::SignalDramaToExit()` as it exits.

12.3.3 Possible Flaws

12.3.3.1 UFACE Case.

What happens if in the case above (where DRAMA has been told to exit) `drama::Task::RunDrama()` loop wakes up after the signal and before the `drama::thread::TUface` destructor has run. It will run `RunDramaExitNotifier::RunDramaHasExited()`, which will wake up any thread waiting on DRAMA messages.

Only flaw here is the lack of joining child threads.

Note that it is up to the application to join the UFACE thread.

Note – the user's sub-class of drama::Task can implement `RunDramaExitNotifier::JoinThreads()` to do this.

12.3.3.2 Action Case

So presume the thread is running and blocked waiting for A DRAMA message. If during this, `drama::Task::RunDrama()` is told to exit, then what happens.

`RunDrama()` will invoke `RunDramaExitNotifier::RunDramaHasExited()` on the action object. This will cause all waiting threads in the action to be woken, with the wait calls throwing an exception. Presume for the moment there is only the one thread involved (the action thread itself).

The action thread will now complete with an exception.

The `drama::thread::TAction` class does re-implement `RunDramaExitNotifier::JoinThreads()` such as to join the action thead. But this does not deal with any child threads of the action thread. The user's sub-class class of drama::Task can implement `RunDramaExitNotifier::JoinThreads()` to do this before invoking the `TAction` implementation.