D R A M A

The Anglo Australian Observatory's distributed instrumentation control system.

Tony Farrell
Anglo-Australian Observatory
Last update 31/5/99 8:07 AM

INCOMPLETE DRAFT   INCOMPLETE DRAFT

DRAMA

Published by the Anglo-Australian Observatory
P.O Box 296
Epping N.S.W 2121
Australia

Contents at a Glance

Table of Contents

Table of Figures

Table of Tables

Table of Examples

Part 1 - DRAMA Basics

1. DRAMA Introduction

Overview

DRAMA is an environment for writing distributed real time systems. It provides a C language API, support libraries, development tools and user interface development tools. DRAMA provides a consistent look and feel to a networked system which can run on various diverse architectures and on all systems provides crisp efficient and reliable communications tools.

This book examines the implementation of DRAMA systems and the various facilities available in DRAMA. It intended to be a solid introduction to programmers wishing to write DRAMA programs.

Quick Intro

DRAMA was written by the Anglo Australian Observatory (AAO) computer group. The AAO is the operator of two large optical telescopes located in New South Wales, Australia, although DRAMA is not specific to telescope operations.

DRAMA has similarities to the Starlink ADAM environment (Starlink is the U.K. Astronomy Computing Support Organisation), but is written entirely in C and has been ported to VMS (both VAX and Alpha), various flavours of Unix (SunOS, Solaris, OSF, Linux[1]) and the VxWorks real-time kernel. A port to the WIN32 interface (For Windows 95 and Windows NT) is well advanced. Ports to other POSIX like systems should be easy. This document assumes no knowledge of ADAM except when interfaces between DRAMA and ADAM are discussed.

The basic unit in DRAMA is the TASK. A DRAMA Task is normally implemented as a separate process within a multi-process operating system. A Task may send or receive messages of various types to other tasks.

The most fundamental message type is the "OBEY" message An associated name specifies the name of a user defined "Action" (or command) that the task will perform. The simplest DRAMA task will set up relationships between "Action" names and C routines. It then enters a loop where messages are received and dispatched to the routine that is the current "handler" for that action. The job of the implementator of a DRAMA task is normally to provide the action routines

---

[1] For the purposes of this document, Linux is identical to any other form of Unix unless otherwise noted.

A DRAMA task may have Parametersassociated with it which may be read or written by other tasks. These parameters provide what is in effect a set of globally accessible variables. They are used to maintain details about a task that may be of interest to other task. We have "GET" and "SET" messages which get and set the value of a named parameter. We also have "MONITOR" messages which allow a task to be notified if the value of a parameter in another task changes. Task parameters may be hierarchical structures of considerable complexity (although they are often simple scalar values).

In addition to an action or parameter name, a DRAMA message may contain an optional user defined structure of any complexity. This provides a very flexible way of transferring information around a DRAMA system, from simple integers to complex images.

User interfaces may be written using X-Windows Xt based widgets (Such as Motif) or using Tcl/Tk.

A typical DRAMA system will use Unix Windows or VMS machines to run user interfaces and CPU intensive operations while VxWorks based VME systems connected via ethernet run the real-time instrumentation. In addition, the real-time components an often be simulated on the VMS or Unix machines for testing purposes.

The Tasking Model

As mentioned above, the basic concept in DRAMA is the "TASK". A "TASK" is an active software object that responds to and initiates messages. The object is normally implemented as a separate process in a multiple process operating system, such as UNIX, VMS or VxWorks.

The *"Fixed Part"* is that part of a task common to all tasks. It access to all the facilities provided by DRAMA. The *"Application Part"* is that part provided by the programmer to implement his specific application. Figure 1.1 shows broad the design of a DRAMA task.

```
            ┌──────────────────┐
            │      DRAMA       │
            │   (fixed part)   │
            └──────────────────┘
               ↕            ↕
    ┌──────────────────┐    ↕
    │   Application    │    ↕
    └──────────────────┘    ↕
               ↕            ↕
          ┌──────────────────────┐
          │  Operating System    │
          └──────────────────────┘
                    ↕
          ┌──────────────────────┐
          │      Hardware        │
          └──────────────────────┘
```
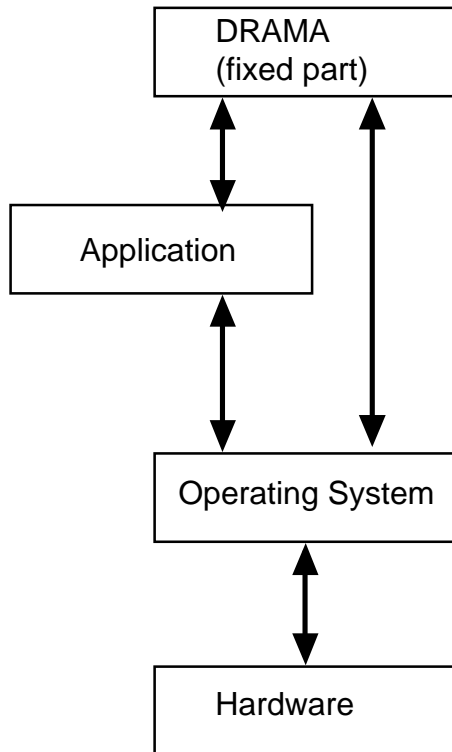
Figure 1.1 DRAMA Task Structure

There are a couple of points to note here. First, the "Application" part is that part written by the task programmer. Second, the application does not need to talk directly to the operating system – if it only uses DRAMA calls, it will be source code portable to any system on which DRAMA runs. But, any application responsible for operating hardware will probably need to access the operating system directly.

The task design has been driven by several different requirements-

- A task responds to messages with/without arguments that may be sent from a number of different sources. The message results in an *"action"* routine being invoked within the task to respond to the message. This action outline is the part supplied by the application programmer. An action has a *"name"* by which it is known outside the task.

- The task sends a completion message to the sender of the original message when the action finishes. This should happen only when the thing initiated by the message has completed. I.e., if the message initiates the moving of a mechanism, the action completes when the mechanism has completed moving to the required position, not when the move has been initiated.

- To avoid tying up the CPU, an action should not poll unless unavoidable. Instead it should block to wait for incoming messages.

- It should be possible to accept additional messages in the same context as the original message, for example abort messages.

- To make resource contention easier to handle a task should be able to handle multiple actions simultaneously within the same process context. (This could be called a form of multi-threading although the threads must control their time splicing explicitly). For example when controlling an `RS232` port it would be dangerous to allow multiple processes to write at will to the port. It is easier and safer to send all messages intended for that port to one process which can then manage access to the `RS232` port.

- The tasking system should not rely on specialized operating system techniques which may dramatically restrict portability. We currently require that it be ported to VAX/VMS, UNIX, Microsoft Windows and VxWorks. As long an underlying message system can be provided, it should be possible to run it on a machine.

- A task should be able to send messages to other tasks and user interfaces should be able to be written without internal knowledge of the tasking system.

- It should be possible for messages, other than completion messages, to be sent to the initiator of an action. For example, it should be possible to output informational and error messages .

- The sending of messages should not cause a task to block waiting only for a response from the target task. This would stop the task accepting messages, such as abort messages, from other tasks.

Event Driven Systems

The model which arises from the above requirements is known as an "Event Driven System". This type of system is on in which most or all work is done in response to an "Event". For example, a car is an event driven system. Some of the events are

Event:	Ignition key is turned
Response:	Power turned on. Starter motor operates
Event:	Accelerator pedal is pressed
Response:	More fuel to the engine
Event:	Break pedal pressed
Response:	Break lights on, engage breaks.

Various software systems work in an event driven way. For example

Windowing Systems   The basic events are the direct result of user interactions, such as mouse and keyboard events. Other events may be generated automatically, such as "Expose" events.

Operating Systems   Normally interrupt driven

Networking Systems   Respond to notification of incoming messages and requests to send messages.

Windowing Systems

For many programmers, their introduction to event driven systems was when they first had to write a program for a windowing system.

Most windowing systems work in the same way - X Windows (including Xt, Athena, Widgets, Motif, Openlook etc.), Microsoft Windows, Tcl/Tk, Mac OS - all use the same basic approach. Since DRAMA uses techniques taken from windowing systems and DRAMA applications must often interact with such systems, we will have quick look at the basic technique.

The basic model of Windowing system applications is

- Initialise Packages
- Associate user defined functions with various events
- Initiate creation of windows etc.
- Enter a loop, usually provided by the windowing system. This loop normally responds at a basic level to incoming events, such as keyboard events and mouse movements.
- At appropriate times, the system supplied loop will invoke the "callback" routines specified earlier

The major job of the programmer is to supply the "Callback" functions, which implement the functionally of the application.

A "Callback" function generally has access to information about the event which triggered it.

Generally, the programmer can associate an item of data (often called "Client Data") with events. This allows the same "Callback" function to be used in different cases or for different objects. It can also be used to allow what would otherwise have to be "Global" data items to be passed in a re-entrant fashion to callback routines.

Messaging Systems

One form of event found in most operating systems is the "Message Arrival" event. As a generalisation, all events can be considered "Messages". This leads to an "Event" driven system where the events are "Messages".

Such messages may be generated explicitly or in response to normal system events such as key press and mouse events.

DRAMA is an example of such message event driven systems.

DRAMA systems

A DRAMA system consists of one or more DRAMA tasks distributed as required across one or more machines possibly of significantly different architectures. DRAMA provides a message system to allow the tasks to communicate whilst hiding architecture specific details and at the same time providing very efficient inter task communications. This system design allows you to run tasks on the most appropriate machine – say user interfaces on Unix under X windows or on a PC running Microsoft Windows whilst real time programs are run on a VME real time system running VxWorks. DRAMA systems typically allow tasks to be added and removed as need to support different configurations and new equipment. Figure 1.2 gives an example of such a system.
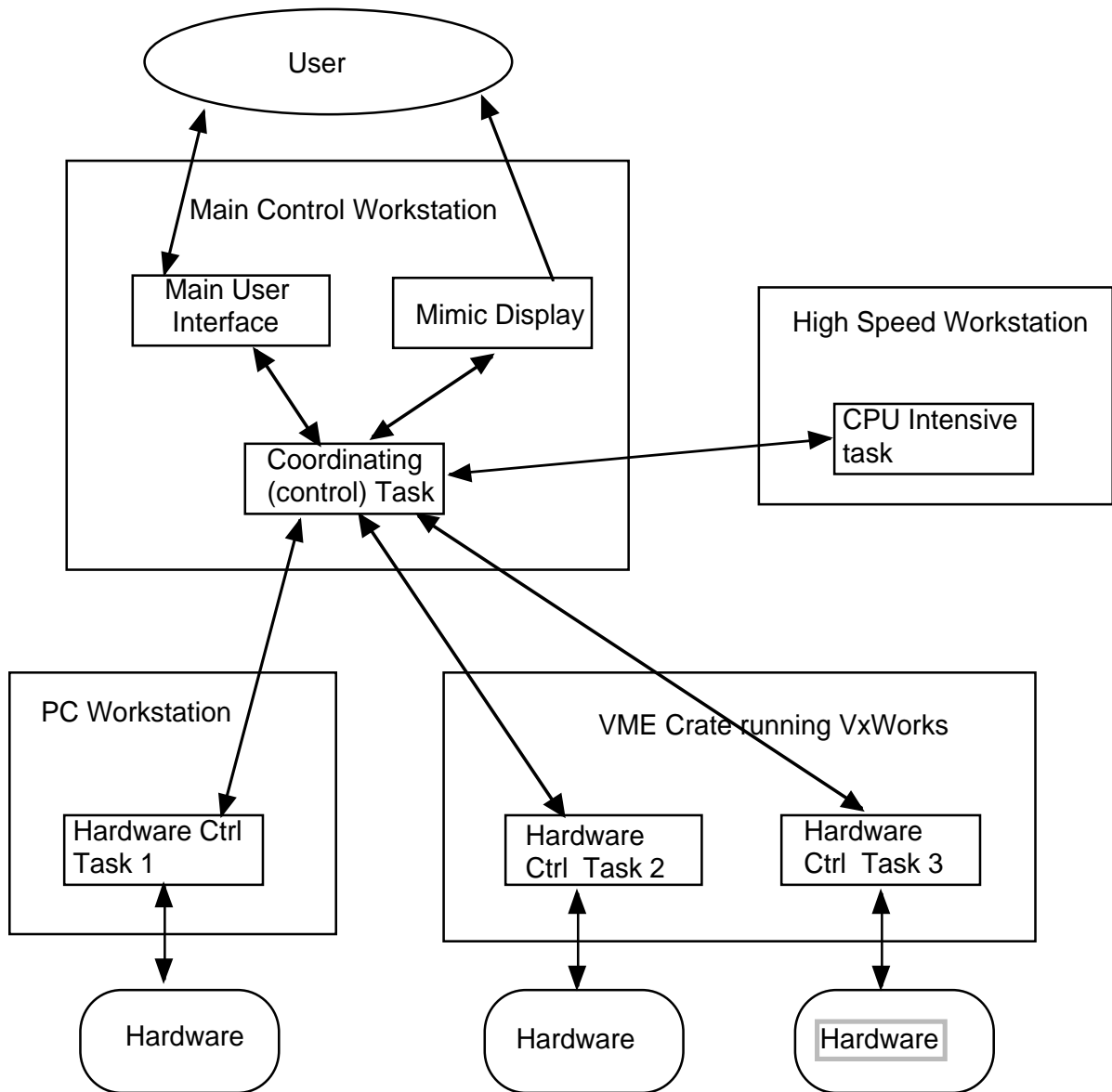
Figure 1.2 A DRAMA System

In figure 1.2, distributed across various machines, DRAMA makes the entire system look like one machine as much as possible. As an example of how this might be useful – the "CPU Intensive task" could be placed anywhere on the network where the required CPU is available.

In addition to providing the message system, DRAMA hides as much as possible the operating system specific interfaces often required by DRAMA task. It does this by providing a common interface. As a result, many DRAMA programs can be built from common source under all supported operating systems and machines.

Documentation   Quick   Intro

A later chapter gives full details on how to find and use the documentation, but here is a quick intro. There is a web site with html and postscript documentation. This can be found at http://www.aao.gov.au/drama/html/dramaintro.html.

If you have a complete Unix DRAMA installation of DRAMA or a DRAMA CD ROM, then you have a local copy of the documentation. The `html` sub-directory contains the html pages, start by opening the `dramaintro.html`. The `docs` sub-directory contains latex source and postscript versions of the latex files. Finally, the `man` sub-directory contains man pages for the sub-routine and command descriptions (also available in `html`). You may need to add this directory to you `MANPATH` environment variable to access these normally.

Where too now?

The rest of this book attempts to introduce you stage by stage to the various features of DRAMA. Part one is about basic tasks, the hardware control tasks in the above examples. Part 2 is about "Control Task" – use to coordinate tasks. Part 3 is about user interface development. Part 4 introduces DRAMA system building and release features. Part 5 contains an assortment of chapters which don't fit anywhere else – to example all the other features found in DRAMA. Finally, Part 6 works through the documentation and how to find, build and install DRAMA.

2. DRAMA "Hello World"

Overview

A tradition in computer science is the "Hello World" program. This is the simplest program you can write in a Language or API which can output the string "Hello World" to the user. DRAMA does have such a program and we shall have a look at the program itself, how to build it and how to run it. Later chapters in this book will expand on this program to demonstrate other DRAMA features.

The code

The main routine of a basic DRAMA program consists of

1   DRAMA Initialisation (`DitsAppInit()`). The arguments to this are the name the task is to be known as, and several other arguments to be described later.
2   Association of message names with handlers (`DitsPutActions()`). It takes as its second argument the address of a structured array. This names the messages and specifies the routines to be invoked when messages of the given names are received. The first argument is the size of the array which can be obtained using the macro routine `DitsNumber()`
3   Invocation of the main loop (`DitsMainLoop()`).
4   Shutdown (`DitsStop()`). We supply to this routine the same name supplied as the first argument to `DitsAppInit()`.   This is done because the name may be used in error messages generated by `DitsStop()` and may not be saved by `DitsAppInit()` if an error occurs in that routine. The value returned by this `DitsStop()` is appropriate for returning to the operating system using `exit()` or return from the main function.

Example 2.1 shows the actual code.

Example 2.1 Hello World

```
#include "DitsTypes.h"          /* Basic Dits types */
#include "DitsSys.h"            /* Initialisation functions */
#include "DitsMsgOut.h"         /* MsgOut   */
#include "DitsFix.h"            /* For DitsPutRequest */


#define BUFSIZE 20000           /* Global buffer size */
#define TASKNAME "DRAMAHELLO"   /* Name of this task */

static void Hello(StatusType *status);/* Prototype of handler */

extern int main(void)
{
    StatusType status = STATUS__OK;
/*
 *  Define the actions supported by this task.
 */
```

```
    static DitsActionDetailsType Actions[] =
        { { Hello, 0, 0, 0, 0, 0, "HELLO"} };

/*
 *  Initialise DRAMA, define the actions we support.
 */
    DitsAppInit(TASKNAME, BUFSIZE,0, 0, &status);
    DitsPutActions(DitsNumber(Actions), Actions, &status);
/*
 *  Enter the Dits main loop.
 */
    DitsMainLoop(&status);
/*
 *  Shutdown, returning an appropriate error code.
 */
    return (DitsStop(TASKNAME, &status));
}
/*
 *  Define the HELLO action handler routine
 */
static void Hello(StatusType * const status)
{
    if (*status != STATUS__OK) return;

    MsgOut(status, "Hello World");
    DitsPutRequest(DITS_REQ_EXIT, status);
}
```

The routine `Hello()` is the message handler routine — the action handler. In this case there are only three lines. We will come back to the first line later. The routine `MsgOut()` is the one which actually outputs the message. The call to `DitsPutRequest()` with the argument `DITS_REQ_EXIT` causes the program to exit once the message is processed.

Include Files

All DRAMA routines are prototyped in include files. The documentation for each routine indicates which include files must be included before using that routine. The include file "`DitsTypes.h`" defines all the general DRAMA types and symbols and should be included first by all programs using DRAMA.

Modified status convention

DRAMA uses a convention for the handling of error conditions known as the "Modified status convention". All routines which take arguments will have a "status" argument. This argument is a pointer to a variable of type `StatusType`. Normally, if the value of the variable is not equal to zero (`STATUS__OK`) then the routine returns immediately. The routine can set the status to a non-zero value when it encounters an error.

In environments without exceptions, this convention makes for simpler code. For example, you can replace the following code

Example   2.2   Error   handling   1

```
if (funct1(a, b, c) != ERROR) {
    if (funct2(d, e, f) != ERROR) {
        if (funct3(g, h, i) != ERROR) {
            if (funct4(a, d, i) != ERROR) {
...
            }
        }
    }
{
```

by something like this

Example   2.3   Error   handling   2

```
funct1(a, b, c, status);
funct2(d, e, f, status);
funct3(g, h, i, status);
funct4(a, d, i, status);
```

A clear advantage in many cases. There are though some situations where this convention can get you into trouble. For example, consider

Example   2.4   Error   Handling   3

```
funct1(&bytes, status)
if ((space = malloc(bytes)) == 0)
    *status = MALLOC__ERROR_STATUS;
funct2(space, status);
```

Consider the case of status being bad when `funct1()` is invoked. `Bytes` may not then be initialised, causing `malloc()` to fail. The failure of `malloc()` causes status to be set, but this overrides the previous value of status. When the error is reported to the user, they will get an incorrect indication of the original error. The solution here is to check the value of status before calling `malloc()` and not call `malloc()` if status is already bad.

If a routine set status to particular values when certain errors occur, then the caller can chose to handle that error. For example, if a routine fails due to not finding a file, the calling routine may choose to create the file rather then regard it as an error. While a failure due to insufficient permission or out of space may be considered fatal and the caller will not attempt to handle these errors.

Additionally, the use of particular bad status values help locate where things have gone wrong. If a particular package were to use unique status values in ever case it set bad status, you can quickly locate in the source code just where a program where a program struck an error.

To support this technique, DRAMA provides support for the generation of unique error codes and the association of text strings with those codes when they are output by user interfaces. This support is provided by the "`messgen`" program and the "Mess" routines, described later.

The DRAMA environment — DRAMASTART

When building and when running DRAMA programs it will be necessary to locate the include files, libraries and executables which make up DRAMA. This is done by a command known generically as "`DRAMASTART`".

The exact form of this command varies depending on the operating system environment. For example

- Under Unix, it is normally "`~drama/dramastart`" where "`~drama`" is the location of DRAMA. If the account "`drama`" does not exist you must replace it with the name of the directory containing DRAMA. Execute this program and follow it by the command "source `$DRAMA/drama.csh`" if your shell is compatible with csh or "`. $DRAMA/drama.sh`" (first character is period) if your shell is compatible with sh.[2]
- For building for VxWorks targets, use "`~drama/dramastart vw68k`" where the "`vw68k`" started for `VxWorks 68K` architecture.
- For VMS, normally "`DRAMASTART`" is sufficient, assuming this symbol has been correctly defined when DRAMA was installed.

The DRAMA environment is almost certainly required to build DRAMA programs. It is normally only necessary at run time to make it easier to locate DRAMA programs.

Building "`dramahello`".

There are three ways to build a simple DRAMA program. The most basic approach is to explicitly invoke the compiler and linker. In this simplest of programs this is quite easy and we will demonstrate how it is done. But as your programs increase in complexity, you will want to use a `Makefile` so we will provide a `Makefile` appropriate for building `dramahello`. An additional technique allows you to create portable build description files which will generate `Makefile`'s appropriate for a given target

Explicit compiling and Linking

The key to compiling and linking a DRAMA program is finding the DRAMA include files and libraries. Assuming your have run the "`DRAMASTART`" command appropriate to your environment then environment variables[3] will exist which can locate these files. In particular, the environment variable "`DITS_DIR`" can be used to locate the libraries

---

[2]Is is quite easy to make the second part of this automatic. For example if you are using csh or tcsh, put the following code in your `.cshrc` file.

```
if ($?DRAMA) then
    source $DRAMA/drama.csh
endif
alias drama ~drama/dramastart
```

[3]Under VMS, logical names are used in place of environment variables.

associated with the DRAMA package known as DITS whilst "DITS_LIB" can be used to locate the libraries.

The basic DRAMA program will use DITS, and hence everything DITS depends on. As a result the DITS package provides assistance which locates everything needed to build such programs. The unix command "$DITS_DIR/dits_cc" will output a string of macro definitions and include file specifications which is appropriate for compiler command lines. Similarly, "$DITS_LIB/dits_link" is appropriate for linking commands. Example 2.5 shows how you use these commands to build dramahello under unix.

Example 2.5 compiling   and linking   dramahello

```
cc -o dramahello dramahello.c `$DITS_LIB/dits_cc` \
    `$DITS_LIB/dits_link` -lm
```

Not that the quotes being used here are backquotes (`` ` ``), not forward quotes ('). Of course you can split this into separate compile and link commands in the normal way.

Under VMS, the approach is a bit different. The logical name "DRAMA_INCLUDES:" is used to locate the include files whist the command "DITS_LINK" is used for linking, as per example 2.6.

Example 2.6 compiling   and linking   under   VMS

```
cc/include=drama_includes dramahello.c
dits_link dramahello
```

A Makefile   for   DRAMAHELLO

The Makefile for the DRAMAHELLO program will of course just reflect the above commands. Example 2.7 shows such a makefile appropriate for using the gcc compiler under solaris 2.

Example 2.7 dramahello   makefile

```
# Set up include directory list.
INCLUDES= `$$DITS_LIB/dits_cc`

# C compiler options
CCOPTIONS = -g -pipe -ansi -Wall

# Command macro
CC = gcc
RM = rm -f
AR = ar rv

CFLAGS = $(CCOPTIONS) $(INCLUDES)

# The list of libraries to link against

LIBS= `$$DITS_LIB/dits_link` -lm

all : dramahello
```

```
dramahello : dramahello.o
    $(RM) $@
    $(CC) -o $@ dramahello.o $(LIBS)

clean ::
    $(RM) dramahello *.BAK *.bak *.o core
    $(RM) errs ,* *~ *.a .emacs_* tags
```

A dmakefile for dramahello

The above `Makefile` is directed towards a simple solaris Unix system. If you want to also build this program under VMS you will need an equivalent (but very different) `descrip.mms` file. If you want to build this program for an embedded system such as VxWorks, you will need another Makefile. You may event need to modify this Makefile for different versions of unix. The result is a maintenance nightmare.

A solution to the problem is required by DRAMA itself, which is ported to many operating systems. The solution provided is to use the X Windows configuration program known as "`imake`". "`imake`" is actually a system as well as a program. The system combines a set of configuration files which describe the abilities of an operating system with a make style description of how to build a particular program. The description file is run through a C pre-processor to generate a Makefile. You do not need to know more about how this works. Just realise that by writing a "`dmakefile`" for your program you can easily generate a Makefile for any target supported by DRAMA. The easiest way to create a `dmakefile` is to copy one which does basically the job you want (which is generally how people create a Makefile). Example 2.8 is a dmakefile for the dramahello program.

Example 2.8 A dmakefile for dramahello

```
#BeginConfig
INCLUDES=DramaIncl IDir(DUL_DIR)
USERCCOPTIONS = AnsiCFull()      /* Enable Full Ansi C */
#EndConfig

NormalCRules()                   /* Include normal c rules    */

DramaProgramTarget(dramahello, Obj(dramahello) ,,,)
```

To generate a `Makefile` for this dmakefile, execute the command "`dmkmf`". After executing this command, you will find a Makefile in the directory. If you wish to generate a descrip.mms file for a VMS system, you need to execute "`dmkmf -t vaxvms`" for a VAX or "`dmkmf -t alphavms`" for alpha VMS machine. You need to execute this command on a Unix machine with DRAMA installed.

Note that although `dmakefiles` are used extensively by DRAMA itself, you need not use them yourself and they are only recommended where you require portability of your DRAMA software to machines with drastically different build procedures eg. VMS and VxWorks.

Running   DRAMAHELLO

Running DRAMAHELLO is quite simple. Under Unix, just run it in the background. (`"./dramahello &"`). Under VMS again just run it (`"run disk:[directory spec]dramahello"`), normally in a spawned sub-process.

Under VxWorks and other, you have several options and details are somewhat environment specific. Basically you need to load the program into VxWorks memory along with the DRAMA infrastructure you will need to run it. I normally do this by first loading the VxWorks version of "DRAMASTART" (`"l d </dramalocation/drama.vw68k"`). I then load the drama libraries object file (`"dld $DITS_DIR/libdits.o"`) and then the program object file (`"dld dramahello.O"`). You can then run dramahello using the VxWorks `"sp"` command (`"sp dramahello"`, assuming you have changed the entry point from `main()` to `dramahello()`).

DITSCMD

"DITSCMD" is a simple program for sending messages to a DRAMA task. It allows you to test simple programs and can be used in scripts running DRAMA programs. We can use it to test the DRAMAHELLO program.

Under Unix, just type "`ditscmd DRAMAHELLO HELLO`".

VMS is very similar. Type "`ditscmd "DRAMAHELLO" "HELLO"`" Note the use of quotes about the arguments to "ditscmd". This is because otherwise VMS will convert these strings to lower case. Since DRAMA task and command names are case sensitive, we need to prevent this.

VxWorks gives you two possibilities. You can just type "`ditscmd "DRAMAHELLO HELLO"`" which runs ditscmd as part of the VxWorks shell. Note that all the arguments to ditscmd are passed as one string.

Alternatively, and somewhat safer is to run ditscmd in an independent task with "`sp ditscmd, "DRAMAHELLO HELLO"`" Note the comma between the command name (ditscmd) and its arguments. This is because both are arguments to "sp", and VxWorks requires arguments to be separated by a comma.

3. DRAMA sub-systems

Overview

The core of DRAMA consists of a number of software packages, known as sub-systems. These sub-systems are documented independently and several of these can be used stand alone. Generally the names of routines and commands found in a package start with a prefix related to the sub-system name, making it easy to find the documentation. The following sub-systems are provided.

MESS — The Message Code system

The MESS package provides a portable way of generating and using message (error) codes. These message codes associate text with an integer in a similar way to the C errno variable values and VMS message codes. We often talk about error codes interchangeably with message codes.

ERS — The Error reporting system.

ERS is the second part of the error reporting system. It provides routines for reporting textual error messages to the user in a highly controlled way. An important part of this is a facility for delayed error reporting. This allows lower level code to report basic error information while the upper level code can add context information or chose not to report the error.

For example, a higher level routine may ask a lower level routine to open a file. If this file does not exist the lower level routine will report such a message and return with bad status. The upper level routine may decide to fail on this error, after adding details about why it was opening the file, or it may instead want to ignore the open failure by creating the file. In the later case, it annuls the error messages reported by the lower level code.

SDS — The Self defining Data System.

SDS allows complex self defining data structures to be built and then transferred as byte streams. It hides details of converting basic data formats (integer byte order, floating point format etc.) as data is transferred between machines.

SDS structures can contain data of any complexity and size, images for example. The Self defining nature means programs can look into a structure to determine its details without extra information needing to be transferred or built into the program.

SDS structures are well suited to moving small or large amounts of data about in heterogenous computer networks. These structures can be sent anywhere a byte steam can be (sent in messages, written to files etc.)

The ARG library is also part of SDS. This library provides routines for creating and accessing SDS structures to be used as command arguments. The resulting package provides simpler access to many common SDS operations.

IMP — The Interprocess Message Protocol.

IMP is the underlying DRAMA message protocol. It will move data between two processes on the same machine as fast as possible whilst providing almost seamless communication between programs on different machines. There is no absolute limit to the size of messages which can be sent and it has been use to send 170Mb images between machines. It is also responsible for reliability issues such as ensuring notification of programs and computers failing. Additionally, it provides a remote task loading facility.

The IMP sub-system is normally hidden from the application programmer by the upper levels of the DRAMA software.

DITS — The Distributed Instrumentation Tasking System.

The DITS sub-system draws the above sub-systems together to implement the basic structure of a DRAMA task. It provides all the basic facilities of DRAMA such a message sending and receiving. It will attempt to provide the most flexible and efficient access to these facilities possible, at the cost of simplicity at times. Other packages make use of some of these facilities to provide simpler interfaces to the most common operations.

Two other related libraries are part of the DITS sub-system. The SDP implements a simple parameter system built largely on SDS. Details of its use will be given later in this book.

The DUI library provides a wrap around of the rather complex facilities provided by DITS for implementing user interfaces. Almost all DRAMA user interface programs are written using this package instead of the underlying DITS routines.

DUL - The DRAMA Utilities Library

The DUL library provides wrap arounds of various common operations in the lower levels. Operations such as sending messages to other tasks are presented in a simpler (but less flexible or less efficient) interface.

The DCPP library is also part of this sub-system. DCPP implements a C++ interface to the messaging operations in DITS.

GIT — The Generic Instrumentation Library

The poorly named GIT library combines two functions. The first is support for the AAO's generic instrumentation software. This allows us to build tasks with consistent interfaces in an object-oriented way. A task "inherits" basic implementations of a set of standard commands. These functions can be used outside the AAO but often a local style will prevail, in which case they can be used as an example of one approach to writing DRAMA software.

The GIT library also provides various utilities functions. These functions really belong in DUL but this library predates that one.

DTCL — The DRAMA TCL Interface.

The DTCL sub-system provides an interface to the TCL[i] scripting language and it's Windowing extension Tk. TCL/Tk is the easiest way to build user interfaces and test scripts in DRAMA. The two systems are well matched.

Philosophy

The structure of the DRAMA sub-systems was designed to provide reliability, easy maintenance and efficiency. It provides core routines which implement basic operations in a flexible and efficient way.

As a result of these quests, it is sometimes found that the interfaces to the basic operations do not match well to the common operations they are used for. For example, the operation to send a message to a task allows multiple messages to be sent at the same time. The requirements of sorting out responses from the multiple messages sent make the interface more complex then the case of sending a single message normally requires. As a result we also provide routines to implement the most common sequences of operations in a simple way. These routines may not be the most efficient or flexible way of doing things but will suffice in most cases.

Whilst we have provided almost all of the core operations which will ever be provided, more interfaces at the higher levels will be provided as experience tells us what is required.

4. SDS — Self defining Data Structures

Overview

The SDS sub-system is fully documented in its own manual[ii] (see http://www.aao.gov.au/drama/doc/ps/sds.ps). We will provide an abbreviated introduction to SDS and its use in DRAMA so as to provide a foundation for future chapters.

SDS structures are used in DRAMA to represent any structure which

- May be read or written by different a task or program from the one that created it. This allows the destination program to be run transparently on a different machine, using different bytes order and floating point formats.

- May have its structure changed dynamically or where the structure varies depending on the situation.

SDS is used in all DRAMA messages and is used by many DRAMA applications as a file format. It is also used by some DRAMA applications to manage dynamically changing structures such as menus.

While used heavily by DRAMA, SDS is independent of DRAMA and is often used in non DRAMA programs.

SDS Formats

There are two SDS formats, When you create SDS structures, you create them in SDS's "Internal format". The details of this format are hidden from the user and accessed only using the supplied routines. This format allows structures to be created and manipulated

To move SDS structures about, you use the "External Format". The external format is a well defined representation of SDS structures in a byte stream. The structure of these items cannot be modified once created although you can change the values. The external format allows SDS structures to be moved anywhere a byte stream can go.

SDS Structures

An SDS structure consists of named items. The name of an item cannot exceed 16 ASCII characters in length. Individual items may be a structure, a scalar item or an array of either structures or scalars.

There are scalar types defined to represent 8, 16, 32 and 64 bit integers, both signed and unsigned, normal length and double length floating point values. An additional scalar type is used for characters.

SDS arrays may have up to 7 dimensions with no restriction on the size of each dimension.

Using SDS

All access to SDS items, either structured or scalar, is via an SDS id. The id is a simple scaler item which can be passed to a routine or program efficiently. A new SDS id is returned when you create an SDS item. In addition, many other functions create SDS ids. Examples here include the functions used to find named items in a structure and those used to index into structured arrays. As a result many SDS ids may point to various parts of the one SDS structure.

SDS structures are created using the `SdsNew()` function. Example 4.1 shows how it can be used to create a simple structure.

Example 4.1 Creating SDS structures

```
#include <stdio.h>
#include "status.h"
#include "sds.h"

int main(void)

{
    SdsIdType topid;        /* Top level identifier  */
    SdsIdType id1;          /* Identifier of first component  */
    SdsIdType id2;          /* Identifier of second component  */
    SdsIdType id3;          /* Identifier of third component  */
    unsigned long dims[2];       /*   Array dimensions    */
    long status;            /* Inherited status variable  */
/*
 * Initialise status variable
 */
    status = STATUS__OK;
/*
 * Create the top level object
 */
    SdsNew(0, "Top", 0, NULL, SDS_STRUCT, 0, NULL,
        &topid, &status);
/*
 * Create the first component - a primitive scalar integer
 */
    SdsNew(topid, "Comp1", 0, NULL, SDS_INT, 0, NULL,
        &id1, &status);
/*
 * Create the second component - a two dimensional double array
 */
    dims[0] = 10;
    dims[1] = 20;
    SdsNew(topid, "Comp2", 0, NULL, SDS_DOUBLE, 2, dims,
        &id2, &status);
/*
 * Create the third component - a structure array - also
 * illustrate the setting of the extra information field
 */
```

```
    dims[0] = 4;
    SdsNew(topid, "Comp3", 17 ,"A Structure Array",
        SDS_STRUCT, 1, dims, &id3, &status);
/*
 * Check everything is OK
 */
    if (status != STATUS__OK)
        printf("Error creating structure - %ld\n", status);

    return(0);
}
```

The top level item of this structure is "Top". The id in the variable topid refers to it. Note that when Top was created, zero was passed as the first argument of the call to SdsNew(). Doing this indicates to SdsNew() that you want to create a new top-level item. The resulting id is then passed as the first argument of the subsequent calls to create the child items. The first item created in this structure is a simple integer item. The second item created is an array of 32 bit integers (or the nearest representation on the platform in question).

The third item created is a structure array. You can then create the individual items in this array, (which may have different structures) or you can fill this array with a copies of a given item. See the SDS manual for full details of the arguments to SdsNew().

In calls to SdsNew(), the following SDS Types are available

Table 4.1 SDS Types and C equivalents.

SDS Type	Details	C equivalent type
SDS_BYTE	used to represent signed byte size integers.	signed char
SDS_UBYTE	used to represent unsigned byte size integers.	unsigned char
SDS_SHORT	16 bit signed integers	short
SDS_USHORT	16 bit unsigned integers	unsigned short
SDS_INT	32 bit signed integers	INT32
SDS_UINT	32 bit unsigned integers	UINT32
SDS_I64	64 bit signed integers	INT64
SDS_UI64	64 bit unsigned integers	UINT64
SDS_FLOAT	single length floating point	float
SDS_DOUBLE	Double length floating point	double
SDS_CHAR	Use to represent byte sized characters	char

Note the difference between byte sized integers and the character type. The later allows for conversion between different character sets and accounts for different sign conventions for "char" types under different C compilers. Also note the use of special types for the 32 and 64 bit integers. These items are typedef-ed to the appropriate type for the compiler.

Having created a structure, we need to be able put data into it and extract it again. We put data using the `SdsPut()` function. You can put data into individual items or use an equivalent C structure to put the entire structure in one operation. An equivalent C structure is one with an equivalent set of item types in the same order. Names of items in the C structure are not significant, just the layou. You use `SdsGet()` to retrieve the values from a structure. Consider example 4.2.

Example 4.2 Getting and Putting SDS structures

```
#include <stdio.h>
#include "status.h"
#include "sds.h"
/*
 * Define a C structure containing 4 items of different types
 */
typedef struct {
    char c1;
    double d1;
    INT32 i1;
    float f1;
} block;

extern int main(void)
{
    StatusType status;
    SdsIdType topid, id1, id2, id3, id4;
    unsigned long actlen;
    block block1 = {'Q', 1.23456789, 9999, 3.1415926};
    block block2;
/*
 * Create an SDS structure equivalent to the C structure
 */
    status = STATUS__OK;
    SdsNew(0, "test", 0, NULL, SDS_STRUCT, 0, NULL,
            &topid, &status);
    SdsNew(topid, "charl", 0, NULL, SDS_CHAR, 0, NULL,
            &id1, &status);
    SdsNew(topid, "doublel", 0, NULL, SDS_DOUBLE, 0, NULL,
            &id2, &status);
    SdsNew(topid, "intl", 0, NULL, SDS_INT, 0, NULL,
            &id3, &status);
    SdsNew(topid, "floatl", 0, NULL, SDS_FLOAT, 0, NULL,
            &id4, &status);
/*
 *  Write the C structure (block1) into the SDS structure
 */
    SdsPut(topid, sizeof(block),0, &block1, &status);
/*
 *  Read from the SDS structure back into the C structure block2
 */
    SdsGet(topid, sizeof(block),0, &block2, &actlen, &status);
/*
 *  Print contents of block2
 */
    printf(" %c %g %ld %f \n", block2.c1, block2.d1,
        block2.i1, block2.f1);
```

```
      return(0);
}
```

In this example we create an SDS structure equivalent to the C structure type "`block`". We then write the contents of one of these blocks to the SDS structure using `SdsPut()`. We retrieve it into a second structure using `SdsGet()` and then print the results.

Deleting structures and Freeing ids

Internal SDS structures should be deleted when you are finished with them. This retrieves all the allocated resources (memory) associated with them. For this you use the function `SdsDelete()`.

Example 4.3 Deleting SDS structures

```
SdsDelete(id, status)
```

This call will delete the structure and all child structures. You can also delete part of a structure by specifying the id of the part you wish to delete.

SDS ids also have resources associated with them and should be free-ed when you are finished with them. This is done using `SdsFreeId()`.

Example 4.4 Freeing SDS ids

```
SdsFreeId(id, status)
```

Normally a call to `SdsDelete()` is followed by a call to `SdsFreeId()` but most calls to `SdsFreeId()` are not preceded by `SdsDelete()`. This is because you will often have many ids pointing to various parts of the same SDS structure. Each of these ids need to be free-ed.

Navigation

SDS provides several functions to navigate around SDS structures. To find a named item in a SDS structure, use `SdsFind`.

Example 4.5 Finding named items

```
SdsFind(parent_id, name, &id, status)
```

In arrays of scalar items you can use `SdsGet()` to get anything from individual items up to the entire array. When dealing with arrays of structures, you need to use `SdsCell()`, which returns the SDS id of a specified array cell.

Example 4.6 Finding array cells

```
SdsCell(parent_id, nindices, indices, &id, status)
```

Additionally, you can step through the members of a structure using `SdsIndex()`.

Example 4.7 Structure members by Index

```
SdsIndex(parent_id, index, &id, status);
```

Note that all these calls return new SDS ids which will need to be free-ed when you are finished with them.

To find out details of an item, use `SdsInfo()`. This returns the name, type and dimensions of an SDS item.

Example 4.8 Structure members by Index

```
SdsInfo(id, name, &code, &ndims, dims, status);
```

External representation

To write an SDS structure to a buffer in it's external format, use `SdsExport()`. The buffer must at least the size returned by `SdsSize()` for the structure. Exporting is the operation used to allow an SDS structure to be moved in a byte stream.

To import an SDS structure from an byte buffer into SDS internal format, use `SdsImport()`. This routine creates a new internal item of the same structure layout and containing the same data. It returns the id of this structure. You would normally only do this if you want to modify the structure's layout. If all you want to do is read or write the data of the structure, then you can use `SdsAccess()` which returns an SDS id which allows you to access the structure in the buffer directly. No internal item is created in this case - the result is known as an External item.

Example 4.9 Exporting and Importing

```
SdsSize(id, &size, status);
SdsExport(id, length, &buffer, status);
SdsImport(buffer, &id, status);
SdsAccess(buffer, &id, status);
```

You should remember that not all operations are possible on external items. Nothing that modifies the structure format is allowed.

Other SDS routines

Several other SDS routines are available. `SdsCopy()` allows you to make a copy of an item. It returns the id of the new item which is always an internal item (the source may be external).

`SdsInsert()` is used to put a new item into structure while `SdsInsertCell()` similarly allows you to insert a cell into an array of structures.

Example 4.10 Other SDS routines

```
SdsCopy(id, &newid, status);
SdsInsert(parent_id, id, status);
SdsInsertCell(array_id, nindicies, indices, id, status);
```

In addition to these routines, several other routines are available which are implemented on top of the above routines. These are know as the SDS utility routines. They also are prefixed by "`SDS`".

General utility routines

To assist in debugging programs, a simple utility routine can be used to write the contents of a structure to the standard output device. This is the routine `SdsList()`. The result is often abbreviated to avoid flooding the output device with data but gives a good indication of the structure of the item and its contents.

Example 4.11   SdsList

```
/* First replace the printf line in example 4.2 with */
    SdsList(topid,&status);

/* Then run the program.   The output is now */
test                    Struct
  char1                 Char   "Q"
  double1               Double 1.23456789
  int1                  Int    9999
  float1                Float  3.14159
```

The routine `SdsFindByPath()` is similar to `SdsFind`, but allows a reference to an item at any depth using a special name format which separates component names by a period. `SdsFind()` only allows references to items at the next level down in the specified item. Note that the use of a period as a separator is purely a convention which is not enforced by SDS. To ensure your structures work with `SdsFindByPath()` don't use periods in your item names.

Example 4.12   SdsFindByPath

```
/* Replace the printf line in example 4.2 with */
    SdsFindByPath(topid, "double1", &id1, &status);
    SdsList(id1,&status);

/* Then run the program.  The output is now */

double1                 Double 1.23456789

```

SDS File I/O

As mentioned earlier, SDS's external representation is a byte stream that can be written to a file using normal file I/O. You can do this yourself but in general you will want to use the utility routines provided by SDS. The routine `SdsWrite()` will write a SDS structure to a file. The routine `SdsRead()` will read a structure from file. The later routines returns the id of an external structure. If you want to modify this structure's structure, then you will need to copy it to get an internal SDS item. Since `SdsRead()` needs to allocate a buffer to read the file into, you should release this buffer when you are

finished with it by calling `SdsReadFree()`. This is normally done just before calling `SdsFreeId`.

Example 4.13   SDS file   I/O

```
/* Reading a Sds structure from a file */
    SdsIdType id=0;
    SdsRead("file.sds", &id, status);

    /* Use the id - now it refers to an external SDS structure*/
    …

    /* Now clean up */
    SdsReadFree(id, status);
    SdsFreeId(id, status);

    /* Writing an Sds structure to a file */
    /* Create the structure */
    …

    /* Write to a file */
    SdsWrite(id, "file.sds", status);
```

The SDS Compiler

Example 4.2 demonstrated the simple mapping between SDS structures and C structures. It is easy to define an SDS structure which can be used to store a given simple C structure. But once the C structure you wish to represent becomes more complicated, it becomes tedious to define the SDS equivalent. The SDS compiler provides the solution. It will take a string/file defining a C structure and produce the sequence of calls necessary for defining an equivalent SDS structure.

The SDS Compiler has two forms. The first form is provided as a run time function call. You pass a string defining a C structure to the function `SdsCompile()` and it returns the id of an equivalent SDS structure. This provides a simple way of constructing SDS structures on the fly but does have the complication of requiring the compiler (and associated libraries) to be linked with your program.

The second form of the SDS compiler is the utility program "`sdsc`". This program will read from its standard input a C structure definition and write to its standard output a C routine definition. This routine is the sequence of SDS calls required to build an equivalent C structure.

The normal procedure is to create an include file which contains the C structure definition. Any program module which wishes to put or get data from equivalent SDS structure will include this file to get the definition of the C structure. It can then just use `SdsGet()` and `SdsPut()` to move data between the SDS and C structures. This include file is also run though the "`sdsc`" program to generate the source of a C module. This C module can be linked with your program and contains a function that can be

invoked to generate the SDS structure and return its id. You have to specify a name for the function on the command line[4].

Any C structure that is representable by SDS can be "compiled". You can use nested structures and "typedef's" in the normal C way to define complicated structures. For maximum portability, use the C equivalent types defined in table 4.1. Although the compiler will also accept types such as "int", and "long int", these are not portable. (long int is 32 bits (INT32) on some machines and 64 bits (INT64) on others.). You may also use the "enum" C type, but these also have portability problems — a compiler option determines if they are to be represented as 16 bit (short) or 32 bit (INT32) integers.

The sdsc program first runs the C pre-processor over it's include file. This allows the standard processing of the source C include file. As sdsc defines the macro __SDSC__ you can use this macro definition in your source code to distinguish source specific to the sdsc run. There are a number of command line options and these are fully documented in the documentation, but the following are used in the example

-f{functionname}	Causes sdsc to generate the entire C function which generates the SDS structure. The argument is the name to be given to the function.
-t{type}	By default, the first variable which is declared in the input file is used to generate the SDS structure. If this argument is specified, there is an explicit declaration of a structure of the specified type.
-N{name}	If -t is used, then this name is given to the created structure.
-D{name[=var]}	Allows you to define C pre-processor macros.

Please note that due to problems running C pre-processors, this feature is not necessarily supported under non-Unix operating systems (VMS and MS Windows).

Example 4.14 shows how to use sdsc.

Example 4.14 SDS Compiler

```
/* Here is a C include file, named struct.h"
typedef struct {
    int a;
    int b[10];
} part1;

typedef struct {
    char name[10];
    part1 value;
} my_struct;
```

---

[4] Originally, the sdsc program only generated the body of a C function. You had to include this body within a C function definition. This removed the need to specify details on the command line but produced a strange result, particularly under some debuggers. The old style still works but is depreciated.

```
/* I ran the sdsc program on this as follows */
sdsc -fcreate_struct -tmy_struct -NTheStruct struct.h struct.c

/* The resulting "struct.c" file is

/*
 * This is a routine for defining a SDS structure based on a C
language
 * declaration.  It has been generated by the SDS compiler.
 * The outer level structure is of type "_anon0002" (struct.h:9)
 *
 * Generated at Mon May 17 08:20:06 1999
 */
#define NULL 0
#include "sds.h"
extern SdsIdType create_struct (StatusType *status)
{
    /* Definition of structure "TheStruct" of type "_anon0002"
       at level 0 (struct.h:9)*/
    SdsIdType tid0 = 0; /* SDS id for structure*/
    unsigned long dims[1]; /* Dimension array for array elements*/
    SdsIdType id = 0; /* Primitive element sds id */
    if (*status != STATUS__OK) return(0);

    SdsNew(0,"TheStruct", 0 , NULL, SDS_STRUCT, 0, NULL,
           &tid0, status);
    dims[0] = 10;
    SdsNew(tid0, "name", 0, NULL, SDS_CHAR, 1 , dims, &id, status);
    SdsFreeId(id,status);
    {
        /* Definition of structure "value" of type
           "_anon0001" at level 1 (struct.h:4)*/
        SdsIdType tid1 = 0; /* SDS id for structure*/
        SdsNew(tid0,"value", 0 , NULL, SDS_STRUCT, 0, NULL,
               &tid1, status);
        SdsNew(tid1, "a", 0, NULL, SDS_INT, 0 , NULL,
               &id, status);
        SdsFreeId(id,status);
        dims[0] = 10;
        SdsNew(tid1, "b", 0, NULL, SDS_INT, 1 , dims, &id,
               status);
        SdsFreeId(id,status);
        SdsFreeId(tid1,status);
        /* End of structure definition at level 1 */

    }
    /* End of structure definition at level 0 */

    return(tid0);
}



/* I can now create a structure of the given format using a
   call to create_create().  I an use SdsPut() and SdsGet()
   to insert and extract data from that SDS structure to a
```

```
   C structure of type my_struct.
*/
```

SDS Utility Programs

In addition to the stand alone version of the SDS compiler, several other SDS utility programs are provided. The program "`sdslist`" is equivalent to the function `SdsList()` for SDS structures stored in files. You specify a list of files containing SDS structure and the contents of these files are listed to the standard output device. Similarly, the program "`sdslistpath`" accepts a period separated path to a sub-structure in the same format as that accepted by the routine `SdsFindByPath()`.

A GUI interface to examining SDS structures is also available. The program "`sdsexam`" takes the name of a file and will display windows which can be used to examine the structure. It provides a better way to examine the complete details of complicated SDS structures at the cost of requiring an X windows display and the full DRAMA release built with Tcl/Tk. (The other utility programs are also available as part of the stand alone SDS release.).

The program "`sdsdump`" dumps all or part of an SDS structure in a file, in a similar fashion to `sdslist`, but where as `sdslist` will truncate its output to avoid screen overflow, `sdsdump` outputs the entire structure.

The program "`sdspoke`" allows you to set the data values in an SDS structure store in a file.

If you have built SDS or DRAMA with support for Starlink enabled, then two additional programs are available. "`sds2hds`" and "`hds2sds`" allow conversion between structures stored in Starlink's HDS file format and SDS structures. Note that for both HDS and SDS structures, the format provides a way of representing named structures but do not enforce ideas of what structures of given names represent. This is up to the applications using the structures.

The ARG routines

The ARG routines are provided as part of SDS. They were intended to ease the implementation of programs using SDS structures to pass command arguments.   They have in practice proved useful in other cases where similar SDS structures are accessed.

The ARG routines assume an SDS structure contains a set of named scalar items, with one exception. The exception is one dimensional arrays of characters, which the ARG routines treat as a character string. Other then this, complicated SDS structures are not handled.

The ARG "Get" routines return the value of the named item within a SDS structure the id of which is specified. The ARG "Put" routines put a value into a named item within an SDS structure, creating the item if it does not already exist. Thus the "Put" routines can be used to create a structure which can be read using the "Get" routines.

A major feature of ARG is automatic type conversion by the "Get" routines. Values being fetched are converted to the type requested if this is possible. The following conversions are attempted.

Table 4.2 ARG type conversions.

String	$\Leftrightarrow$	Integer
String	$\Leftrightarrow$	Floating Point
Integer	$\Leftrightarrow$	Floating Point

All type sizes representable in SDS are supported. Integer types may be signed or unsigned. Conversions from floating point formats to integer formats will cause rounding. If the source represents a value too large to be represented in the destination type, then an error is returned.

Use `ArgNew()` to create an empty top-level SDS structure for use by the "Put" routines. The "Get" routines are of the form `ArgGetx()` where x is one of [c i u s us 64 u64 f d string] representing C types "char", "long", "unsigned long", "short", "unsigned short", "INT64", "UINT64", "float", "double" and "char *". The "Put" routines have an equivalent format — `ArgPutx()`. For `ArgGetString()` an extra argument is used to pass the size of the buffer for the returned string.

Example 4.16 The ARG routines

```c
#include <stdio.h>
#include "status.h"
#include "sds.h"
#include "arg.h"

extern int main(void)
{

    SdsIdType id;          /* SDS identifier */
    StatusType status;     /* Inherited Status Variable */
    status = STATUS__OK;
/*
 *  Create a structure
 */
    ArgNew(&id, &status);
/*
 *  Create an integer component
 */
    ArgPuti(id, "Comp1", 123, &status);
/*
 *  Create a float component
 */
    ArgPutf(id, "Comp2", 3.141592, &status);
```

```
/*
 *  Create a string component
 */
    ArgPutString(id, "Comp3", "This is a string", &status);
/*
 *  List the results
 */
    SdsList(id, &status);
    return(0);
}
```

The "Get" and "Put" routines will operate on any SDS structure, not just one generated by `ArgNew()`. `ArgNew()` is provided to simplify the interface to `SdsNew()` for creating the simple SDS top level structure required. In addition, the SDS item created by the ARG routines can accessed using all the normal SDS functions.

SDS Summary

This chapter has provided an introduction to SDS sufficient to get you started and to understand the SDS code used in DRAMA examples in following chapters. For full details of SDS and complete routine descriptions, please refer to the SDS manual.

5. DRAMA Obey Messages

Overview

Of the DRAMA messages which can be received by a DRAMA task, the fundamental one is the Obey message type. Almost all DRAMA programs will handle Obey messages.

This chapter examines how the application programmer writes code to handle DRAMA Obey messages.

Each Obey message has a name associated with it. This name indicates what the Task should do in response to the message. A Task implements an Action (or command) in response to an Obey message.

Arranging   handling   of Obey messages

A task tells DRAMA the names of the Obey messages it will support and associates a routine with each name by calling `DitsPutActions()`. The specified routine is invoked automatically by the DRAMA internals when a message of the given name is received.

We repeat below the code from the "Hello World" in example 2.1.

Example   5.1   Hello   World

```
#include "DitsTypes.h"          /* Basic Dits types */
#include "DitsSys.h"            /* Initialisation functions */
#include "DitsMsgOut.h"         /* MsgOut    */
#include "DitsFix.h"            /* For DitsPutRequest */


#define BUFSIZE 20000           /* Global buffer size */
#define TASKNAME "DRAMAHELLO"   /* Name of this task */

static void Hello(StatusType *status);/* Prototype of handler */

extern int main(void)
{
    StatusType status = STATUS__OK;
/*
 *  Define the actions supported by this task.
 */
    static DitsActionDetailsType Actions[] =
        { { Hello, 0, 0, 0, 0, 0, "HELLO"} };

/*
 *  Initialise DRAMA, define the actions we support.
 */
    DitsAppInit(TASKNAME, BUFSIZE,0, 0, &status);
    DitsPutActions(DitsNumber(Actions),Actions, &status);
/*
```

```
 *   Enter the Dits main loop.
 */
    DitsMainLoop(&status);
/*
 *   Shutdown, returning an appropriate error code.
 */
    return (DitsStop(TASKNAME, &status));
}
/*
 *   Define the HELLO action handler routine
 */
static void Hello(StatusType * const status)
{
    if (*status != STATUS__OK) return;

    MsgOut(status, "Hello World");
    DitsPutRequest(DITS_REQ_EXIT, status);
}
```

`DitsPutActions()` takes the size of and address of an array of structures of type `DitsActionsDetailsType`. This structure has 7 components, the first of which is the address of the routine to be invoked when an obey message of the specified name is received whilst the last is a character string which is the name of the obey message. All other components are optional and can be specified as 0 (zero or the null pointer, depending on the argument[5]). The function `DitsNumber()` used as part of the first argument to `DitsPutActions()` is actually a macro which given an array variable, will return the number of elements in the array.

In this example, only one action is supported, named "HELLO". Action names are case sensitive[6] so the names "HELLO", "Hello" and "hello" would represent different actions. The routine which will be invoked for a message with this name is the "Hello" routine. This is prototyped before being specified as a routine taking one argument of type pointer to `StatusType` and returning `void`. This routine is known as the "Action Routine" for the "Action" (or command) named "HELLO".

In the example, the Action Routine definition is very simple. The first thing to note is the slightly different argument typing — the pointer is marked as "`const`". This ensures we don't assign to the pointer instead of what is pointing to and catches a common error where the pointer is set to zero instead of the value it is pointing to. You can give the prototype declaration the same format but it is unnecessary in ANSI C and needlessly

---

[5]The ANSI C standard specifies that 0 (zero) can always be used to represent a null pointer. It is automatically converted to a null pointer of the required type. Since the commonly used `NULL` macro is often cast to either "void *" or "char *" it creates some portability problems - so DRAMA sticks to using 0 where ever a null pointer is required.

[6]If your actions may have to be invoked by an ADAM task, you should stick using upper case action names since ADAM only supports upper case names.

exposes details of the function definition at the prototyping stage[7]. DRAMA uses this style though out. This is a style issue only but I felt I should explain the difference.

The first line which does anything is the check of status. The routine obeys the convention of returning immediately if status is bad on entry. In practice, status is never bad on entry to an action routine, but you should code it this way as you may later decide to call this routine from a different location.

Next we consider the following line.

Example 5.2 Basic message output

```
    MsgOut(status, "Hello World");
```

Here we output our "Hello World" message to the user. Of special note here is the position of the "`status`" variable. The `()` takes as it's second argument, a "`printf()`" style format string, followed by a variable numbers of arguments to the format. In this example, no such formatting is done so there are no extra arguments. A requirement of having a variable number of arguments is that the fixed arguments be first, thus "`status`" must be one of the first two arguments. It is before the format string to keep the format string with its arguments. Each message output by `MsgOut()` will be output on a separate line — there is no need to use "`\n`" and it is ignored if you do so.

The only other thing left to do is to cause the program to exit

Example 5.3 Program exit

```
    DitsPutRequest(DITS_REQ_EXIT, status);
```

The routine `DitsPutRequest()` is used to communicate to the DRAMA fixed part what should happen when the action handler returns. Various requests are available, the request `DITS_REQ_EXIT` tells the DRAMA fixed part that the program should exit (actually, all that really happens is that `DitsMainLoop()` returns, the actual exiting of the program is done using the return from the main function.) If no call is made to `DitsPutRequest()` a default request is made by DRAMA with a code of `DITS_REQ_END` This indicates to the fixed part that the action is completed. In this case, the action may then be invoked again by a new message.

That is all there is to it. When a user interface program, such as `ditscmd.` is used to send an obey message named "`HELLO`" to this task, the message is output and the program exits.

Real world requirements

We now know how to build the simplest DRAMA program, but in order to do real work of the type DRAMA is designed for, we need to introduce various other facilities.

---

[7]You should note that some compilers don't handle this correctly. They force the prototype and definition to be the same. The DEC Alpha-OSF compiler is one of these.

In the real time systems which make up the majority of DRAMA systems, the implementation of an action may be time consuming. For example, if the above task is to move a motor on reception of the Obey message, it will have to wait while the motor moves to ensure the job is done correctly. This is normally done using interrupt driven techniques to avoid tying up the CPU and will often require a timeout to be implemented to catch mechanical problems.

An additional requirement in real world systems is the ability to kill an action cleanly or to respond to requests for changes in the process of an action. In the above example, the motor may take several minutes to move to a new position, if we send the command accidentally we will want to stop the operation if physically possible.

Rescheduling

In order to support these requirements in a portable way, DRAMA implements a technique known as "Rescheduling". This works as follows. An action starts the operation it implements (say "move filter wheel") and "requests" that instead of ending when the action routine returns, it be "Rescheduled" as a result of an interrupt, timeout or subsequent message.

The invocation of the action upon the requested event is known a "Rescheduling" the action. An action can be rescheduled multiple times allowing complicated sequences to be implemented by an action.

We will examine the implementation of the simplest form of rescheduling — implementing a delay. This technique can be used by itself to implement simple delays or in conjunction with say interrupt driven rescheduling to implement a timeout

Example 5.4 Timer Rescheduling

```
static void Hello(StatusType * const status)
{
    if (*status != STATUS__OK) return;

    if (DitsGetSeq() == 0)
    {
        DitsDeltaTimeType delay;
        MsgOut(status, "First entry");
/*
 *      First entry, create a delay structure and enable it.
 */
        DitsDeltaTime(5, 0, &delay);
        DitsPutDelay(&delay, status);
/*
 *      Request a wait.
 */
        DitsPutRequest(DITS_REQ_WAIT, status);
    }
    else
    {
```

```
/*
 *      Second entry, cause the task to exit.
 */

        MsgOut(status, "Second entry");
        DitsPutRequest(DITS_REQ_EXIT, status);
    }
}
```

The first new function called here is `DitsGetSeq()`. This function returns an integer which is known as the "*sequence count*". This is zero the first time the routine is invoked in the context of the one Obey message and is incremented on each reschedule event for this action. In this example, we use the sequence counter to separate the code for the first and subsequent entry to this routine.

On the first entry, we set up the timer based reschedule. This is a two part procedure, the first part being to create a timer value. You use the routine `DitsDeltaTime()` to create a timer value. The arguments here are the number of seconds and microseconds and the address of the time value, of type `DitsDeltaTimeType`. Once created a timer value can be used multiple times. We use this one in the call to `DitsPutDelay()`, which indicates to the DRAMA fixed part that this action is to get a timer reschedule message the given interval after this actions returns. This separation of creating the timer value and putting it makes for more efficient operation where a timer value is being used repeatedly. A higher level routine, `GitPutDelay()`, does both operations in one call.

Having created and put the timer value, we must indicate to the DRAMA fixed part that we want this action to be rescheduled on expiry of the timer. If we don't do this, the timer is ignored. We do this with the call to `DitsPutRequest()` with the value `DITS_REQ_WAIT`. We then return from this routine.

DRAMA then arranges for this routine to be invoked again when the timer expires. This time `DitsGetSeq()` returns one and the second part of the code is invoked where we just output a message and request the program exit.

Rescheduling to a different routine

The use of `DitsGetSeq()` and an if-then-else or switch statement sequence to separate the different stages of an action can complicate coding of an action. Normally, the code to handle the next part of an action will be either exactly the same or entirely different. In languages like C where you are encouraged to write a lot of small functions, it makes more sense to be able to specify the routine to handle the next stage, instead of using `DitsGetSeq()` to determine what to do when the next stage occurs. This would remove one layer of coding and make it easier to write module and actions handlers that can be used by different programs.

The function `DitsPutObeyHandler()` is used to change the routine to be invoked when an the action is rescheduled. The following example shows the previous example rewritten to use this to split the handling of the different stages into different functions.

Example 5.5 Different handler rescheduling

```
static void HelloStage2(StatusType * status);

static void Hello(StatusType * const status)
{
    DitsDeltaTimeType delay;
    if (*status != STATUS__OK) return;

    MsgOut(status, "First entry");
    DitsDeltaTime(5, 0, &delay);
    DitsPutDelay(&delay, status);
/*
 *  Change the handler and then request a wait.
 */
    DitsPutObeyHandler(HelloStage2,status);
    DitsPutRequest(DITS_REQ_WAIT, status);
}

static void HelloStage2(StatusType * const status)
{
    MsgOut(status, "Second entry");
    DitsPutRequest(DITS_REQ_EXIT, status);
}
```

For complicated actions, the resulting programming style of is often neater, clearer, more modular and more efficient. When the action is started a second time, the handler is automatically reset to the original routine.

Multiple Actions

A given task may respond to action messages of more then one name. For example, a simple task which moves and monitors a filter wheel may respond to actions "INITIALISE", "MOVE_FILTER" and "EXIT". This ability allows the implementation of applications which must maintain context across such operations. Typical real applications at the AAO have five to fifty different actions and where they access a physical instrument or device, will be the only task to access that instrument directly.

This way of building DRAMA tasks is a matter of style. You may prefer to build a lot of simple tasks to access the one physical device, each with one action or a small number of actions.

To specify multiple actions, you add more entries to the action array specifying details of each action. For example, we may prefer to split the HELLO action itself from the shutdown of the task by providing an EXIT action. This will allow us to send the HELLO action to the same task multiple times. The "Actions" array now looks like

Example 5.6 Multiple Actions - actions array

```
static DitsActionDetailsType Actions[] =
    { { Hello, 0, 0, 0, 0, 0, "HELLO"},
      { Exit,  0, 0, 0, 0, 0, "EXIT"} };
```

The routine `Exit()` will be invoked when the `EXIT` action is received. The `Hello()` can now be modified such that it no longer causes the program to exit.

Example 5.7 Multiple Actions - routines

```
static void Hello(StatusType * const status)
{
    if (*status != STATUS__OK) return;

    if (DitsGetSeq() == 0)
    {
        DitsDeltaTimeType delay;
        MsgOut(status, "First entry");
/*
 *      First entry, create a delay structure and enable it.
 */
        DitsDeltaTime(5, 0, &delay);
        DitsPutDelay(&delay, status);
/*
 *      Request a wait.
 */
        DitsPutRequest(DITS_REQ_WAIT, status);
    }
    else
    {
/*
 *      Second entry.  Action completes.
 */
        MsgOut(status, "Second entry");
        DitsPutRequest(DITS_REQ_END, status);/* Optional */
    }
}

static void Exit(StatusType * const status)
{
    if (*status != STATUS__OK) return;
    DitsPutRequest(DITS_REQ_EXIT, status);
}
```

The `Hello()` routine now puts a request of `DITS_REQ_END` on its second entry instead of `DITS_REQ_EXIT`. This request indicates that the action has completed but the task is left running. When an action completes, the task which sent the obey message gets an action completion message. This request is the default request so putting it is optional. The `DITS_REQ_EXIT` just does a `DITS_REQ_END` before telling the task to exit from `DitsMainLoop()`.

Simultaneous Actions

Rescheduling allows multiple actions to be running at the same time. The code of the two actions is not running at the same time, but whilst one action is waiting for a reschedule event another may be running. This relies on the application programmer coding his application to use the rescheduling facilities to wait for events instead of just doing simple polling or blocking. This technique is known as "co-operative multi-tasking" and is the most portable way to implement multi-tasking applications. For example, an application can start two filter wheels moving at the same time in response to

two different messages from the user. Consider the following Actions array used with the code from example 5.7 above.

Example   5.8   Simultaneous   Actions

```
static DitsActionDetailsType Actions[] =
    { { Hello, 0, 0, 0, 0, 0, "HELLO1"},
      { Hello, 0, 0, 0, 0, 0, "HELLO2"},
      { Exit,  0, 0, 0, 0, 0, "EXIT"} };
```

Here we have replaced the "HELLO" action by two actions, "HELLO1" and "HELLO2". For my convenience only I have continued to specify the one routine to be invoked in each case. Your application would normally require you use different action routines for different actions. By extending the delay being created by `DitsDeltaTime()` to about ten seconds, it is easy for you to use the `ditscmd` program from different terminals to send first `HELLO1` and then `HELLO2`. The messages output will show the switching of the two actions.

Simultaneous   Actions   of the same name

In the above example, I created actions of different names to demonstrate the simultaneous actions feature. By default, DRAMA only allows one action of the same name to be active at the same time. If you send a second obey to a task whist one of the same name is still active, the second obey will be rejected with a message indicating the action is already active.

This style of operation is normally preferred when physical devices are being controlled. For example, you would not normally want the user to try to move the same filter wheel multiple times, nor does it make sense to allow multiple Initialisation actions to be running at the same time.

It is possible for you to specify that multiple actions of the same name be allowed using the flag `DITS_M_SPAWNABLE` as the fourth item in an action definition entry as shown below

Example 5.9 Simultaneous   Actions   - Same Name

```
static DitsActionDetailsType Actions[] =
    { { Hello, 0, 0, DITS_M_SPAWNABLE, 0, 0, "HELLO"},
      { Exit,  0, 0, 0, 0, 0, "EXIT"} };
```

In this example, an unlimited number of actions named "HELLO" may be active at the same time — only limited by the amount of memory available.

If you wish to limit the number of actions of this name, specify a routine as the fifth item in the entry. This routine is of type `DitsSpawnCheckRoutineType` and will be invoked each time an action of this name is to be started. It takes two arguments, a user specified item (the sixth item in the entry) and the address of a status variable. Its return type is void. If it returns with status ok, the action is allowed to start, otherwise the message is rejected with the value of status being supplied as the reason for the rejection.

The use of a routine instead of a hard limit allows the numbers of actions to be limited depending on the dynamic state of the program.

Beware that if the flag is an "`or`" of multiple flags including `DITS_M_ACT_INFO` or `DITS_M_ACT_CLEANUP` then the sixth item is the address of a structure instead of the spawn check data. This structure's type is `DitsActInfoType` and its first item is the spawn check data item. See the `DitsPutActions()` routine description for more details.

Also beware that it is harder to "Kick" spawn able actions. Kicking is explained in the next chapter.

6. Kick Messages, More on Rescheduling

Overview

The second DRAMA message we will examining is the Kick message. This message allows communication from outside a task to an active rescheduling Action in a task. The most common use of the Kick message is to cancel or abort an Action but they may be used for any other reason which requires communication with an active action, such as altering the progress of an action. For example, you may have an action named "EXPOSE" which implements a Camera device exposure. An exposure can take a long time and you may want to cancel the exposure or alter the exposure time once it is running. The Kick message is the appropriate message to use.

Basic Kicking

To add the ability to kick an action, you need only specify a routine to be invoked when a kick message arrives for that action. This is the second item in the action map supplied to `DitsPutActions`. The following example is our rescheduling variation of Hello world (separate EXIT action) with Kicking of the "HELLO" action supported using the `KickHello()` routine.

Example 6.1 Kicking HELLO

```
#include "DitsTypes.h"          /* Basic Dits types */
#include "DitsSys.h"            /* Initialisation functions */
#include "DitsMsgOut.h"         /* MsgOut    */
#include "DitsFix.h"            /* For DitsPutRequest */


#define BUFSIZE 20000           /* Global buffer size */
#define TASKNAME "DRAMAHELLO"   /* Name of this task */

static void Hello(StatusType *status);/* Prototype of handlers*/
static void KickHello(StatusType * status);
static void Exit(StatusType *status);

extern int main(void)
{
    StatusType status = STATUS__OK;
/*
 *  Define the actions supported by this task.
 */
    static DitsActionDetailsType Actions[] =
        { { Hello, KickHello, 0, 0, 0, 0, "HELLO"},
          { Exit,  0, 0, 0, 0, 0, "EXIT"} };
/*
 *  Initialise DRAMA, define the actions we support.
 */
    DitsAppInit(TASKNAME, BUFSIZE,0, 0, &status);
    DitsPutActions(DitsNumber(Actions),Actions, &status);
/*
 *  Enter the Dits main loop.
 */
```

```
    DitsMainLoop(&status);
/*
 *  Shutdown, returning an appropriate error code.
 */
    return (DitsStop(TASKNAME, &status));
}
/*
 *  Define the HELLO action handler routine
 */
static void Hello(StatusType * const status)
{
    if (*status != STATUS__OK) return;

    if (DitsGetSeq() == 0)
    {
        DitsDeltaTimeType delay;
        MsgOut(status, "First entry");
        DitsDeltaTime(10, 0, &delay);
        DitsPutDelay(&delay, status);
        DitsPutRequest(DITS_REQ_WAIT, status);

    }
    else
    {
        MsgOut(status, "Second entry");
        DitsPutRequest(DITS_REQ_END, status);
    }
}
/*
 * Handle kick of the HELLO action by ending the action.
 */
static void KickHello(StatusType * const status)
{
    DitsPutRequest(DITS_REQ_END, status);
}


static void Exit(StatusType * const status)
{
    DitsPutRequest(DITS_REQ_EXIT, status);
}
```

To run this example using the "ditscmd" command, send the Obey message from one session in the normal way. To send the Kick message, run a second copy of ditscmd, with the same arguments but with the flag "-k" between the command and the taskname argument. This command indicates that a Kick message should be sent instead of the default of an Obey message.

In this example the Kick routine causes the action to end immediately. The other possibilities are described later.

You can also dynamically set and change the kick handler to be in effect for future reschedules of the action using DitsPutKickHandler(), which is similar to DitsPutObeyHandler().

Strands of control.

DRAMA tends to talk about the "Obey" strand of control and the "Kick" strand of control. The Obey strands are the normal entries to Action handlers triggered by Obey messages and reschedule events. The Kick strand of control is an entry to the Kick routine. Although both strands of control sometimes look similar, you never reschedule the Kick strand of control — instead the Kick routine can change the rescheduling of the Obey strand. The Kick strand of control is always triggered by reception of a Kick message. Entries to both the Obey and Kick strand cause the sequence count (returned by `DitsGetSeq()`) to be incremented.

You can find out what strand of control you are running in by calling `DitsGetContext()`, which returns `DITS_CTX_OBEY` for an obey strand and `DITS_CTX_KICKED` for a Kick strand. An additional possible return value, `DITS_CTX_UFACE`, is explained in the section on User Interfaces and is not normally of interest to application programmers.

Rescheduling options

We have already seen a couple of the possibilities as the first argument to `DitsPutRequest()`. We will now examine all the possibilities.

DITS_REQ_END   This is the default request when an Obey routine is invoked, both on the first entry and on subsequent reschedules. It is not the default when routines are invoked by a Kick message. This request indicates the action has completed and a completion message is sent to the originator of the Obey message

DITS_REQ_EXIT   As per `DITS_REQ_END`, but after the completion message is sent, the DRAMA fixed part will exit, causing `DitsMainLoop()` to return.

DITS_REQ_STAGE   A reschedule event will be queued for this action immediately it returns to the fixed part. The result is that the action is invoked again as soon as any other outstanding messages are handled. You can use this request to break up your action code in such a way as to allow Kick messages and other Obey messages to be received.

DITS_REQ_WAIT   A reschedule event will be queued after expiry of a timer. The timer must be put independently using `DitsPutDelay()`. `DITS_REQ_STAGE` is equivalent to `DITS_REQ_WAIT` if the timer delay for the later request is zero seconds.

DITS_REQ_SLEEP   The action is put to sleep. This is like the `DITS_REQ_WAIT` request except that the putting of a timer delay is optional. It is normally used to put the action to sleep to wait for interrupt event messages (See `DitsSignalByName()`) and Kick messages.

DITS_REQ_MESSAGE   Wait for a reschedule event triggered by the reception of a message directed towards this action. Such a message is in response to a message

initiated previously by this action. See Chapters on control tasks for more information. A timeout can also be put using `DitsPutDelay()`.

If you find that both `DITS_REQ_SLEEP` and `DITS_REQ_MESSAGE` apply, use the later.

The one other case to consider is the default behaviour for entries to Kick routines. If no request is put in a Kick routine, then the Obey strand of control is not affected by the Kick message. If status is bad when the routine returns, then it is considered that the Kick message has been rejected and the status returned is considered the reason for the rejection. If status is not bad, then the Kick is considered to have succeeded, but simply does not want to effect the rescheduling, possibly just updating a value to be examined on the next Obey reschedule or it might output progress information.

7. DRAMA Message Arguments

Overview

DRAMA messages can have arguments. These arguments are in the form of a single SDS structure which is associated with the message and is known as the Argument Structure. There may be only one such structure associated with each message but since it is an SDS structure it can be of any complexity and it is easy to simulate a set of arguments.

Handling Arguments.

The argument structure associated with an Obey or Kick message can be accessed with the function `DitsGetArgument`. This function returns the SDS id of this argument structure (type `SdsIdType`), if one was supplied. It returns zero (0) if no argument structure was associated with the message. (Zero is always an invalid SDS id.)

You do not need to free this id or delete the structure, it is cleaned up by DRAMA itself after your action handler routine returns. But you should beware that the argument structure is not maintained across rescheduling of an action so you need to copy it using `SdsCopy()` if you want to keep it. Arguments are normally external SDS structures so you cannot modify the structure format although you can modify it's contents.

User Command Arguments.

The format of the SDS arguments sent with messages is up to the applications involved. You can send large images if required. In completed applications, the user interface is often a specialised windowing based application which will validate user input (making use of other DRAMA facilities) before building the argument structure required by the application task the message is being sent to. For example, the user may use the user interface to specify an image file. The user interface can then read the file itself and specify the image as an argument to an Obey message sent to the application. The underlying application task here need not and should not be bothering with things such as keyword associations, defaults, prompting for values in error etc, these are better left to the user interface. These behaviours were essential to older command line based applications and nothing in DRAMA prevents this approach, but it is not supported in the DRAMA core.

Note, the above policy does not remove the requirement for the underlying application to check ranges and types of arguments etc., but it would normally handle problems here by simply rejecting the message (by setting status bad before returning). This serves as a check on the user interface.

A problem with this approach is that it makes it hard to run simple programs and test a system as it is being built as it is impossible to implement common user interface programs. A convention is required to allow the building of common user interface programs otherwise applications won't know how to treat the argument structures they receive.

The convention is that structures generated by standard user interfaces (such as ditscmd) are built using the ARG library. Each element in such structures are simple scalar values, with the exception that character strings are represented using character arrays. The names of items will be "Argument1", "Argument2" … "Argument$n$" where "n" is the number of arguments. This of course restricts you to sending commands where the arguments fit this approach, but there is probably nothing which can be done about Actions which require more complicated argument structures, such a images — specialised user interface needs to be built (this is not hard, as we will see later).

As you might expect, the "ditscmd" program uses this approach. Any extra arguments after the action name are put into such an argument structure. The following example shows how to handle such an argument in application.

Example 7.1 Accessing Arguments

```
#include "DitsTypes.h"
#include "DitsSys.h"
#include "DitsMsgOut.h"
#include "DitsFix.h"
#include "sds.h"            /* For SDS stuff */
#include "arg.h"            /* For ARG routines */


#define BUFSIZE 20000
#define TASKNAME "DRAMAHELLO"

static void Hello(StatusType *status);
static void Exit(StatusType *status);

extern int main(void)
{
    StatusType status = STATUS__OK;
/*
 *  Define the actions supported by this task.
 */
    static DitsActionDetailsType Actions[] =
        { { Hello, 0, 0, 0, 0, 0, "HELLO"},
          { Exit,  0, 0, 0, 0, 0, "EXIT"} };
/*
 *  Initialise DRAMA, define the actions we support.
 */
    DitsAppInit(TASKNAME, BUFSIZE, 0, 0, &status);
    DitsPutActions(DitsNumber(Actions),Actions, &status);
/*
 *  Enter the Dits main loop.
 */
    DitsMainLoop(&status);
/*
 *  Shutdown, returning an appropriate error code.
```

```
 */
    return (DitsStop(TASKNAME, &status));
}

static void Hello(StatusType * const status)
{
    char string[30];
    if (*status != STATUS__OK) return;
/*
 *  Get our argument as a string.
 *  Status will be set bad if there is no argument.
 */
    ArgGetString(DitsGetArgument(),"Argument1",
        sizeof(string),string, status);

    MsgOut(status, "Argument is \"%s\"", string);

    DitsPutRequest(DITS_REQ_END, status);
}

static void Exit(StatusType * const status)
{
    DitsPutRequest(DITS_REQ_EXIT, status);
}
```

The important line is the call to `ArgGetString()`. This call returns to the call the value of the item named "Argument1" in the SDS structure referred to by the id returned by `DitsGetArgument()`. The argument concerned is converted to a string if it is not one already. If you want it as say an integer, replace this call by `ArgGeti()`. Such calls will complain if the supplied argument cannot be converted to the requested types. This example just outputs the argument value using `MsgOut()`.

Also of interest here is the `GitArg` set of routines, such as `GitArgGeti()`, which returns an integer. These routines implement range checking and other facilities such as automatic case conversion for string values. The can also be used to fetch arguments by either name or position in the structure, allowing some flexibility in the ordering of arguments where specific user interfaces are used to build argument structures. See the GIT library for more details[iii] (see http://www.aao.gov.au/drama/doc/ps/git_spec_9.ps).

Output Arguments.

In the same way that argument structures can be associated with the messages that start Obey's, arguments can be associated with the messages sent to the parent task when the action has completed. In addition, intermediate messages can be sent with arguments attached.

Arguments associated with the completion of an action are normally used to return some value which is the response of the action. Thus actions may both do work and return values. You construct such arguments using the SDS and/or ARG routines. The association of the SDS structure with the completion message is done using `DitsPutArgument()`. The first argument to this routine is the SDS id of the structure. The second argument to `DitsPutArgument()` indicates what DRAMA

should do with the structure after the message has been sent. Since the completion message with the argument attached is sent after your action routine has completed, the SDS structure must out last your action routine code and must be deleted, if necessary, by DRAMA. This flag has one of the following values

DITS_ARG_DELETE The SDS structure specified is to be deleted after the message is sent and the SDS id freed.

DITS_ARG_COPY   The SDS structure is immediately copied. The copy is deleted after DRAMA has sent the message. This code is useful when you want to continue modifying the SDS structure.

DITS_ARG_NODELETE   The SDS structure is not deleted by DRAMA nor is the SDS id freed. This flag is useful if you want the structure to survive the action completion.

The association of SDS structure with the action completion message only applies if the action completes on the entry which calls `DitsPutArgument()`, rather then rescheduling.

To send an intermediate message, say to indicate progress of an action, use `DitsTrigger()`. This routine's first argument is the SDS id of a structure to be associated with the message. The term "trigger" indicates that the major reason for this routine is to "trigger" wake up the parent of the action. Since this routine operates immediately, no special flags are required and you may delete the SDS structure immediately the routine returns. Note that trigger messages normally require special handling by the parent task. Many such tasks will by default ignore Trigger messages or report an error. To use `DitsTrigger()`, include "`DitsInteraction.h`".

Example 7.2 Output Arguments

```
#include "DitsTypes.h"
#include "DitsSys.h"
#include "DitsMsgOut.h"
#include "DitsFix.h"
#include "DitsInteraction.h"
#include "sds.h"              /* For SDS stuff */
#include "arg.h"              /* For ARG routines */


#define BUFSIZE 20000
#define TASKNAME "DRAMAHELLO"


static void Hello(StatusType *status);
static void Exit(StatusType *status);


extern int main(void)
{
    StatusType status = STATUS__OK;
/*
 *  Define the actions supported by this task.
```

```
 */
    static DitsActionDetailsType Actions[] =
        { { Hello, 0, 0, 0, 0, 0, "HELLO"},
          { Exit,  0, 0, 0, 0, 0, "EXIT"} };
/*
 *  Initialise DRAMA, define the actions we support.
 */
    DitsAppInit(TASKNAME, BUFSIZE, 0, 0, &status);
    DitsPutActions(DitsNumber(Actions),Actions, &status);
/*
 *  Enter the Dits main loop.
 */
    DitsMainLoop(&status);
/*
 *  Shutdown, returning an appropriate error code.
 */
    return (DitsStop(TASKNAME, &status));
}

static void Hello(StatusType * const status)
{
    SdsIdType id;
    if (*status != STATUS__OK) return;
/*
 *  Create an argument structure
 */
    ArgNew(&id, status);
    ArgPuti(id, "RESULT", 1, status);
/*
 *  Trigger the parent action, send this structure
 */
    DitsTrigger(id, status);
/*
 *  Associate the structure with the completion message.
 */
    DitsPutArgument(id, DITS_ARG_DELETE, status);
    DitsPutRequest(DITS_REQ_END, status);
}

static void Exit(StatusType * const status)
{
    DitsPutRequest(DITS_REQ_EXIT, status);
}
```

8.Parameters

Overview

A task may have "Parameters". Parameters are named data items accessible to both the task itself and to other tasks. A task uses Get messages to get the values of parameters belonging to other tasks. It may also use Set messages to set the values of parameters in other tasks. In these messages, the name associated with the message is the name of the parameter involved.

Normally parameters are used for one of two reasons.

- To store configuration information which may need to be changed by other applications. You may for example use parameter values to set a task into debugging mode while testing it.

- To maintain state information about a task which may be required by other tasks. This is by far the most important usage.

Parameter   Systems

The DRAMA core supports the Get and Set messages but does not define a parameter system. The task author is responsible for this, allowing the author to choose the type of the parameter system to be supported or to choose not to have one at all. The author could for example choose to use an external data base system (such as EPICS) as the parameter system, choose to emulate an older parameter system (such as the Starlink ADAM's parameter system) or the author could decide that the application requires the lightest tasks possible and as a result choose to have no parameter system. In the later case, Get and Set messages to the task are rejected.

The parameter system chosen by the author of a task determines how task itself accesses parameter values, normally providing a set of routines which are invoked to get and set the parameters belonging to the task.

Our example programs so far have not provided a parameter system. To provide one, you must tell DRAMA which routines are to be invoked for Get and Set messages. This is done using `DitsAppParamSys()`[8]. This routine both allows you to set the routines involved and an inquire about the existing routines. Full details can be found in the DITS manual[iv] (see http://www.aao.gov.au/drama/doc/ps/dits_spec_5.ps). Normally the parameter system initialisation routine would invoke `DitsAppParamSys()` on your behalf and you need only concern yourself with this routine if implementing or modifying a parameter system.

---

[8]Older code uses `DitsPutParSys()`.   This routine has been superseded b y `DitsAppParamSys()`.

The SDP parameter   system

Although DRAMA does not enforce the use of a single parameter system or indeed require any parameter system at all, it does provide one. The "Simple DITS Parameter" System, SDP, uses SDS to implement a minimal parameter system that has proved to be sufficient in most cases.

SDP stores all parameters in a single SDS structure. Parameter names correspond to top level items in this structure. Since SDS is used, parameters may be structured allowing parameters of any complexity including images.

By convention, simple parameters (scalars and character strings (null terminated single dimensional character arrays) use upper case only names whilst other parameters use mixed case names. This allows older ADAM systems to access the simple parameters.

Example 9.8 shows how to add the SDP parameter system to the DRAMAHELLO program.   The call `SdpInit()` creates the parameter system and calls `DitsPutParSys()`. This should be done after calling `DitsAppInit()` and before calling `DitsMainLoop()`.

Example 8.1 Parameters

```
#include "DitsTypes.h"
#include "DitsSys.h"
#include "DitsMsgOut.h"
#include "DitsFix.h"
#include "Sdp.h"              /* For the parameter system. */
#include "sds.h"             /* For SDS stuff */
#include "arg.h"             /* For ARG routines */


#define BUFSIZE 20000
#define TASKNAME "DRAMAHELLO"

static void Hello(StatusType *status);
static void Exit(StatusType *status);

extern int main(void)
{
    StatusType status = STATUS__OK;
/*
 *  Define the actions supported by this task.
 */
    static DitsActionDetailsType Actions[] =
        { { Hello, 0, 0, 0, 0, 0, "HELLO"},
          { Exit,  0, 0, 0, 0, 0, "EXIT"} };

/*
 *  Parameters supported by this task.
 */
    static INT32 one = 1;
    static double fone = 1.0;
```

```
        static SdpParDefType params[] = {
            { "PARAM1", &one,         SDP_INT },
            { "PARAM2", &fone,        SDP_DOUBLE },
            };
        SdsIdType parsysid = 0;


/*
 *  Initialise DRAMA, define the actions we support.
 */
        DitsAppInit(TASKNAME, BUFSIZE, 0, 0, &status);
        DitsPutActions(DitsNumber(Actions),Actions, &status);
/*
 *  Initialise parameter system.
 */
        SdpInit(&parsysid, &status);
        SdpCreate(parsysid, DitsNumber(params),params, &status);
/*
 *  Enter the Dits main loop.
 */
        DitsMainLoop(&status);
/*
 *  Shutdown, returning an appropriate error code.
 */
        return (DitsStop(TASKNAME, &status));
}

static void Hello(StatusType * const status)
{
        long value;
        if (*status != STATUS__OK) return;
/*
 *  Get our argument as a string.  Status will be set bad
 *  if there is no argument
 */
        ArgGeti(DitsGetArgument(),"Argument1", &value, status);
        MsgOut(status, "Argument is \"%ld\"", value);
/*
 *  Put the value into the parameter PARAM1
 */
        SdpPuti("PARAM1", value, status);

        DitsPutRequest(DITS_REQ_END, status);
}

static void Exit(StatusType * const status)
{
        DitsPutRequest(DITS_REQ_EXIT, status);
}
```

To create parameters you use `SdpCreate()` or `SdpCreateItem()`. The former routine is used to create a set of parameters described by an array of structures of type `SdpParDefType`. This structure has three elements, being the name of the parameter (character string pointer), a pointer to the initial value (`void *`) and the type of the parameter, as per below.

Table 8.1 SDP parameter types.

Type	Details	C Type
SDP_BYTE	used to represent signed byte size integers.	signed char *
SDP_UBYTE	used to represent unsigned byte size integers.	unsigned char *
SDP_SHORT	16 bit signed integers	short *
SDP_USHORT	16 bit unsigned integers	unsigned short *
SDP_INT	32 bit signed integers	INT32 *
SDP_UINT	32 bit unsigned integers	UINT32 *
SDP_I64	64 bit signed integers	INT64 *
SDP_UI64	64 bit unsigned integers	UINT64 *
SDP_FLOAT	single length floating point	float *
SDP_DOUBLE	Double length floating point	double *
SDP_CHAR	Use to represent byte sized characters (not strings)	char *
SDP_STRING	Use to represent character strings.	char *
SDP_SDS	Use to specify structured or array parameters. Construct the SDS structure and put the address of it here.	SdsIdType *

You can see that there is a scalar SDP type equivalent to every SDS scalar type and two extra types, SDP_STRING and SDP_SDS. For each of the scalar types, the default value is specified by specifying the address of an item of the C type from the third column of the table as the second element in the structure.

The SDP_STRING type is used for character string types. The value in this case is the address of a null terminated string. This type is represented using an SDS character array.

The SDP_SDS type can be used to represent any other type using a SDS structure. You specify for the default value the address of an SDS id. The associated SDS structure will be inserted into parameter system. This allows you to make any structure representable by SDS into a parameter — images for example. Probably a better approach is to add such complex parameters at run time, one at a time. To create a single parameter you can use SdpCreateItem(). You specify with this routine the parameter system id and the SDS id of the item to be inserted.

Getting/Putting Scalar and String SDP parameters.

A task which uses SDP as its parameter system may use a sequence of simple calls to get and change the values of scalar parameters. These calls are similar to the ARG series of calls. The above example uses SdpPuti() to change the value of a parameter. It could have used SdpGeti() to get the value of the parameter. These routines are based on

the ARG routines and perform type conversion in the same way, i.e. the value is converted if necessary to the requested type.

Additionally, as per the ARG routines, there are routines to get and put character strings, `SdpGetString()` and `SdpPutString()`.

Note that some older code may use the ARG routines directly, specifying the value returned by `DitsGetParId()` as the first argument. This still works but may not have all the functionally of the SDP routines. In particular, for the Put versions, the parameter monitor system described below, will not work. In addition, use of the ARG routines may cause a compilation warning unless the value of `DitsGetParId()` is cast to a `SdsIdType`.

Getting/Putting Complex SDP parameters.

When getting and putting the values of non-scalar SDP parameters (arrays, complex structures etc.), it is harder to generalise the interface. In addition, for some parameters (such as images) efficiency may become very important - you don't want values copied unnecessarily. Since these parameters are implemented as SDS items, SDP provides support for accessing them directly using SDS. This allows you to use SDS features such as `SdsPointer()` to access items efficiently.

You can use `SdpGetSds()` to get the SDS id of a parameter. Since the SDP parameter system is implemented as one SDS structure, all this routine does is a `SdsFind()` call using the name of the parameter and the id of the parameter system. It returns the resulting SDS id, which should be freed when you are finished with it. Once you have an id to the item, you can use all the normal SDS facilities, `SdsPointer()` to access the data directly, `SdsFind()` to navigate etc. You can even change the structure of the item.

When you change the value such a parameter, you should notify SDP of the change using `SdpUpdate()`. You specify the SDS id of the parameter that you received from `SdpGetSds()`. If you are doing a batch of changes to a given parameter, you need only call `SdpUpdate()` at the end. Note that the requirement to call `SdpUpdate()` does not stop another task from getting the value of the parameter while you are changing it. That task will see the value in its intermediate state (but only if your task reschedules in the middle of the changes). `SdpUpdate()` is used by the parameter monitor system (see below) to indicate that tasks monitoring parameters should be told of the change.

Alternatively, you may change the value of structured parameters using `SdsPutStruct()`. This call takes a parameter name and an SDS id and makes the specified SDS item the new value of the named parameter. This is somewhat less efficient than the combination of `SdpGetSds()` and `SdpUpdate()` but prevents other tasks seeing the intermediate state.

Reserved SDP parameter names

SDP reserves all names starting and ending with an underscore. This allows the implementation of special features fully documented in the `SdpGet()` routine description.

Currently, the following special parameter names are implemented.

`_ALL_`	When specified as the parameter name in a Get message this name causes the entire parameter system to be returned.
`_NAMES_`	When specified as the parameter name in a Get message this name causes the names of the parameters to be returned. The returned value is an SDS structure of type SdpNames. It contains one item which is a 2 dimensional SDS array, representing an array of null terminated character strings being the names of the parameters.
`_LONG_`	Indicates that the parameter name is specified in the SDS structure as well as the value. This allows us to specify parameter names of any length.

Long Parameter Names

The SDP parameter system supports long parameter names in `SET` and `GET` messages. This allows parameter names to be specified using the format accepted by the `SdsFindByPath()` function. This allows you to set and get the values components of structured DRAMA parameters. See the `SdsFindByPath()` function for more details of the naming format `SET` messages involving long parameter names make use of the `_LONG_` special parameter name to allow the parameter name be part of the message argument structure.

To send a `GET` message involving long parameter names, the `MGET` message is used instead of the `GET` message. Once again, the parameter name becomes part of the argument to the message allowing it to be of any length.

Get multiple parameter values

DRAMA itself, rather then `SDP`, supports the concept of getting multiple parameter values in one message rather then having to set multiple `GET` messages. This is done with a `MGET` message, which specifies parameter names using an argument to the message rather then in the message "name" structure.

Read only parameters

In the default way of working, any other task can change the values of a parameter. Sometimes this is undesirable. There are two ways around this depending on the features required. If you wish your entire parameter system to be read only from other task s(all they can do is get the parameter values) invoke the routine `SdpSetReadonly()` after invoking `SdpInit()`.

If on the other hand, you wish read only for some parameters only or you wish to validate the values of parameters, consider specifying your own parameter setting routine using `DitsAppParamSys()`. This set routine will be invoked with the parameter system id, parameter name and SDS id of the new value. You can validate the parameter value however you please before invoking `SdpSet()` to set the parameter value or rejecting it by returning bad status. In effect, you are creating your own parameter system but using SDP to help you implement it and provide consistency.

Parameter Monitoring

One of the most powerful features of DRAMA is the parameter monitor facility. A new message type known as the Monitor message supports this feature.

The basic facility involves two tasks, say task A and task B. Task A tells task B that it is interested in the value of one or more parameters belonging to task B. Task A will then receive a message each time the values of the specified parameters are changed.

Consider for example the case of a user interface (task A) which operates a camera device (task B) which has a shutter. If the shutter state is placed by task B in a parameter, then the user interface can monitor that parameter and display its values on the user interface. The important thing here is that task B need do nothing other then put the state of the shutter in a parameter. All details of the monitor operation are hidden from task B by DRAMA. There may be any number of tasks monitoring any number of task B parameters. In addition, the work required by task A is probably less then it would be otherwise, for example if it explicitly polled for the position of the shutter.

Alternatively, a different type of monitor allows task A to tell task B to send the value of parameter to a third task, say task C. Consider the above example. Task B may be able to make available in a parameter, the images being read from the camera. To avoid replication of code, a common program which can display images may be available. This second type of monitor allows task A to tell task B to forward the value of this parameter to task C. This will be done each time the parameter changes. Various task A/B combinations may use the same task C to display images.

As a further example, task C may maintain in a parameter the details of areas of the image selected by the user. Task A could monitor these parameter values and use them to set windows, specify centroiding regions etc.

The monitor facility has proved very powerful in both user interfaces and in cases where a common control task must monitor the state of other tasks.

Parameter monitoring example

As a demonstration of parameter monitoring, use the program from example 8.1. You will need two terminal sessions. From session 1, start the example program. From session two, enter the following command

Example 8.2 Parameter   monitoring   1

```
ditscmd -p DRAMAHELLO START PARAM1
```

Now go to the session 1 terminal and type

Example 8.3 Parameter   monitoring   2

```
ditscmd DRAMAHELLO HELLO 10
```

This results in the parameter PARAM1 being changed. A message will be received on terminal session 2. Alternatively, you can set the parameter directly from session 1 using

Example 8.4 Parameters

```
ditscmd -p DRAMAHELLO PARAM1 30
```

Parameter monitoring uses a transaction that you set up with the first invocation of ditscmd (the -p option). That message told the DRAMAHELLO task that the ditscmd task was interested in the parameter PARAM1. A message was returned containing the monitor transaction id. This id can be used to change details of the monitor transaction, just as which parameters are monitored and to cancel the transaction. The subsequent invocations of ditscmd changed the parameter value resulting in the messages being sent to the original version of ditscmd[9]. Note, the original version of ditscmd is left running.

To cancel the monitor transaction, you a monitor message with a name of "CANCEL", specfying an argument which is the integer monitor id returned by the "START" message. For example, from session 1 type.

Example 8.5 Canceling   a parameter   Monitor

```
ditscmd -p DRAMAHELLO CANCEL 1
```

The full session one window transcript it.

```
% ./dramahello
% ditscmd DRAMAHELLO HELLO 10
DITSCMD_2642:DRAMAHELLO:Argument is "10"
% ditscmd -p DRAMAHELLO CANCEL 1
```

while the session 2transacript is

```
% ditscmd -p DRAMAHELLO START PARAM1
DITSCMD_263f:Monitor start message from task "DRAMAHELLO". ID = 1.
DITSCMD_263f:Monitor  message  from  task  "DRAMAHELLO".  Parameter
PARAM1 = 10.
```

---

[9]"ditscmd" is run under a different task name each time it is started.   This is required to allow multiple versions of ditscmd to run at the same time.

Once again, `DitsAppParamSys()` is the underlying routine to set up parameter monitoring. In addition to the parameter setting and getting routines, you can specify routines to support parameter monitoring. For a full specification of the parameter monitoring message, please see the `DitsInitiateMessage()` function description.

.

9. Message/Status codes.

Overview

In chapter 2 you were introduced to the DRAMA status convention. You will recall that DRAMA uses a convention for the handling of error conditions known as the "Modified status convention". All routines which take arguments will have a "status" argument. This argument is a pointer to a variable of type `StatusType`. Normally, if the value of the variable is not equal to zero (`STATUS__OK`) then the routine returns immediately. The routine can set the status to a non-zero value when it encounters an error.

Additionally, the use of particular bad status values help locate where things have gone wrong. If a particular package were to use unique status values in every case it set bad status, you can quickly locate in the source code just where a program struck an error.

We will now examine the use of particular bad status codes and their generation.

Status/Error   codes

Error/Status/Message codes (the terms are used interchangeably) are integer values which will fit within a 32 bit signed number. The actual StatusType type will be at least 32 bits in size, but will generally correspond to the size used by Fortran programs for simple integers. This allows compatibility with a large amount of existing Fortran code.

The `StatusType` variable is defined in the include file "`status.h`", which also defines one value, `STATUS__OK`, which is always defined to be zero (for compatibility reasons)

All other error codes should be defined to a value calculated using the equation

```
CODE = 134250496 + 65536 x facility  + 8 x message number +
       severity
```

Here, the *message number* (in the range 1 to 4095) is assigned to the error condition by the author of the subroutine library. The *facility* number (in the range 1 to 2047) allows message numbers to be independent of other systems, assuming the use of an allocation scheme[10]. Facility numbers in the range 1800 to 1899, and the number 2000, may be used by DRAMA software and should be avoided.

The *severity* is used to distinguish between different levels of error. The use of severity is solely a convention, but allows tasks/subroutines to indicate to controlling tasks/routines that an error is not fatal. Severity is a three bit number, with the following values used

---

[10]Software groups associated with the UK astronomy community use facility numbers from a range of facility numbers allocated to each institution by the Starlink ADAM Support Group.

Value	Name	Meaning
0	WARNING	Warning only, invoking code can consider ignoring this failure
1	NORMAL	Can be taken to mean that no error has occurred, but provides information about the result of a function. A leftover from older systems, it is generally not used in DRAMA code.
2	ERROR	A failure has occurred.
3	INFORMATIONAL	Can be taken to mean that no error has occurred, but provides information about the result of a function. A leftover from older systems, it is generally not used in DRAMA code.
4	FATAL/SEVERE	A fatal error has occurred, retrying is not likely to work.

Other values have no meaning. Remember that severity is purely a convention, any status value other then zero is generally taken as indicating an error has occurred, it is up to the application code to implement any special handling of other values.

Why use error codes.

The DRAMA error codes are simple integer values. They can be compared efficiently and passed about the network. At a most basic level, a routine will set status to a particular code when something goes wrong. If all the calling routines simply return on bad status, this bad status will eventually arrive at a level in the program where it can be delivered to the user. Since error codes can be unique to a particular sub-routine, it may be possible to determine exactly where the program failed.

Additionally, the higher levels of the code may be capable of handling particular errors. For example, if a routine which opens a file fails due to the file not existing, the caller may examine the error code and in this case, create the file. The calling routine in this case may be a subroutine call or involving the sending of a message to a another task anywhere on the network.

In summary, the DRAMA error code system allows you to determine where an error came from and how to handle it.

Associating text with error codes.

As mentioned above, error codes may eventually be returned to a level where they must be presented to the user. Integer error codes are of limited use at this point. DRAMA error codes generated in the normal way (described below) have a text string associated with them. Routines are provided to retrieve the text string given the error code, although this relies on the program fetching the string knowing the correct relationship. The text associated with a message is of the form

```
FACILITY-L-IDENT, message-text
```

Where FACILITY is a name associated with the facility number, L is the severity (one of S (Success), I (Informational), W (Warning), E (Error), F (Fatal/Severe)), IDENT is a name associated with the message number. The "message-text" can be any text the author wishes to associate with the error code and normally explains the error. For example

```
DITS-E-FILEOPEN, Failed to open file
```

might indicate a file open failure in a DITS routine.

Creating error codes, Error code include files.

In general, to allow easy use of the integer error codes in C source, one would create a C "#define" macro to represent each message code. A standard exists which says that the macro name is the combination of the facility name string and the message name string, separated by two underscores. For example, the error code the text of which is shown above would be represented by the C macro `DITS__FILEOPEN`.

The normal procedure is to put all the error codes from one facility in to one include file and allow access to the include file by any application which may need to know what the values are.

To generate such include files, DRAMA provides a utility known as messgen — the message code generator. This program takes a file which defines the facility name and number and contains a list of the message names. For example,

Example 9.1 Error code generation

```
.FACILITY DITS,2000/PREFIX=DITS__
!
! these lines are comments
!
.SEVERITY FATAL
!
FILEOPEN <Failed to open file>
TASKDISC <Task disconnected>
MACHLOST <The machine has been lost>
INVMSGLEN <Message length is too small>/WARNING
.END
```

This file is for the facility DITS, named in the first line, as is the facility number. The prefix defines the first part of the macro name. Comment lines start with explanation marks. We specify the default severity using the ".SEVERITY" line. Three message codes are defined, with numbers starting from 1. The generated include file will contain the definitions of the macro's `DITS__FILEOPEN`, `DITS__TASKDISC`, `DITS__MACHLOST` and `DITS__INVMSGLEN`. The last message here uses a different severity — "WARNING". For full details of the format of this file, see the Message Generation Utility document[v] (see http://www.aao.gov.au/drama/doc/ps/mess_6.ps).

Traditionally, the above is put in a file the name of which is something like "dits_err.msg".

To generate the required include file, use the following command

Example 9.2 Running Messgen

```
messgen -c dits_err.msg
```

The file generated where will be named dits_err.h. Other options to messgen allow Pascal, Fortran and Tcl compatible files to be generated and allow the output files specified.

Error code to text associations.

Another option to the messgen command (-t) causes it to generate an additional C include file which sets up a structure containing the text associated with each code. The file generated is suffixed with "_msgt.h". In the above example, the file would be "dits_err_msgt.h". Routines are provided to use these tables to provide translation of error codes to the corresponding text string.

To enable the translation of a particular facility, you must tell the translation functions about it.   The include file generated above will contain a structure named "MessFac_*FACNAME*" (eg., MessFac_DITS in the above example).   Your application initialisation sequence should invoke the routine MessPutFacility() specifying the address of this structure, eg.

Example 9.3 Adding Facilities

```
MessPutFacility(&MessFac_DITS);
```

The translation details of this facility are now available. The basic routine to translate an error code is MessGetMsg(), which takes as arguments the error code, a flags word and text buffer details. Eg.

Example 9.4 Translating message codes

```
char buffer[200];
MessGetMsg(DITS__FILEOPEN, 0, sizeof(buffer), buffer);
```

The flags argument allows you to fetch only parts of the translation, for example, the text only. An additional routine, MessPutFlags(), allows you to set the default mode for this flag.

The above routines are independent of the DITS layer of DRAMA. Within a DRAMA task, you can use a DITS routine with a simpler interface. DitsErrorText() takes a single argument, the error code. It returns the address of a buffer with the text. Subsequent calls will overwrite this buffer and the function cannot be invoked at interrupt or signal context. Eg.

Example 9.5 Use of DitsErrorText

```
MsgOut(0, status, "Translation is %s",
                DitsErrorText(DITS__FILEOPEN);
```

Other MESS functions.

The MESS library provides several other functions which help to handle error codes.   Of particular interest is `MessSeverity()` which returns the severity of an error code. Note that it is possible to change the severity of a error code by masking out the existing severity and or-ing with the desired severity. See i for details.

Largely for consistency, functions are provided to access the Facility number (`MessFacility()`) and message number (`MessNumber()`).

Higher level functions.

Several higher level functions are provided to help in managing the Message code facilities. These features are provided by the DUL library. You can load a message facility file (.msgt_h file) on demand using `DulReadFacility()`. In addition, the routine `DulLoadFacs()`, loads all the standard DRAMA facilities as well as any defined by the environment variable `DRAMA_FACILITIES`. The later is a colon-separated list of message facilitie files to load. The standard higher level DRAMA user interfaces (`xditscmd`, `dtcl`, `dtk`) invoke this routine automatically.

VMS Compatibility.

This error code system is compatible with a subset of the features provided by VMS MESSAGE program and system routines, used to provide exception handling and error messages in Digital's VMS operating system. This compatibility arose due to the origins of DRAMA's predecessor (ADAM) on VAX/VMS machines.

In general, most VMS message error facility files can be compiled by messgen and in addition DRAMA message facility files can be compiled into a VMS object format by the VMS "`MESSAGE`" command. Such object files can be linked with a VMS program to allow translation of the error code by VMS system calls. There may be some facility files which can be translated by one system but not the other, but these are rare.

For consistency, VMS versions of the MESS routines will fall back to operating with the VMS system calls if translations can't be found within the MESS system. This allows DRAMA programs to transparently handle VMS system call error codes using the same features as used for DRAMA error codes.

DRAMA   6 9

10. Control Messages

Overview

The last of the DRAMA message types is the control message. This special type of message is used to allow a task to communicate with the DRAMA fixed part of another task. Currently, there are five message names supported, the DEFAULT message allows a task's default directory to be retrieved or changed whilst the MESSAGE control message allows an external task to translate status codes. The DEBUG message allows you to remotely set the internal DRAMA debug flag. The DUMPPATHS and DUMPTRANS messages cause details of the valid DITS paths and outstanding transactions of the task respectively to be written to `stdout`. The -c option to the `ditscmd` program allows you to send control messages.

DEFAULT control messages

This message allows a task's default directory to be fetched or changed. If an argument is supplied, then it is the new default directory. The current default directory (after any change) is returned regardless of if an argument is supplied.

A task can change the routine which handles this message using the function `DitsPutDefaultHandler()`. This routine allows you to specify a routine to be invoked and a client_data item to pass to it. It will also return the current values, allowing you to chain routines if desired. The routine to be supplied takes a number of arguments, including the `client_data` item, the new directory specification, and a place to return the resulting directory specification. If the new specification is a null string, then it should just return the current value.

On initialisation, DRAMA makes an implicit call to this routine, specifying the routine `DitsDefault()`.

MESSAGE control messages

These messages enable a task to translate the integer error codes returned by a task into text. It is necessary as only the task itself may have available all the message code to text translations required (this have been set up using the routine `MessPutFacility()`). The argument to the message is a character string containing an integer. This integer may be decimal or hexadecimal string acceptable by the C run time library `strtol()` `function`. The return value for the message is the transaction.

DEBUG/DUMPATHS/DUMPTRANSID control messages

The DEBUG messages enable you to remotely enable debugging of a DRAMA task. The argument is converted to an integer using the C run time library `strtol()` function and then passed to the routine `DitsSetDebug()`. If no argument is supplied, "1"

(one) is used. `DitsSetDebug()` enables or disables output of various internal debugging information, dependent on its argument.

The DUMPPATHS message causes details of all paths known to the task to be written to `stdout`. The DUMPTRANSID message causes all details of all outstanding transactions to be written to `stdout`.

11. Error reporting

Overview

Message/Error codes are only the first part of the DRAMA error handling system. They allow an application program to handle errors in what ever way is appropriate whilst providing a basic approach to the generation of textual error messages. It is though quite limited. It does not provide for multiple error messages as might be required to give context to an error report and it does not allow arguments to be added to messages, such as the name of files involved.

ERS is the second part of the error reporting system. It provides routines for reporting textual error messages to the user in a highly controlled way. An important part of this is a facility for delayed error reporting. This allows lower level code to report basic error information while the upper level code can add context information or chose not to report the error.

For example, a higher level routine may ask a lower level routine to open a file. If this file does not exist the lower level routine will report such a message and return with bad status. The upper level routine may decide to fail on this error, after adding details about why it was opening the file, or it may instead want to ignore the open failure by creating the file. In the later case, it annuls the error messages reported by the lower level code.

Reporting   Errors

Most textual error reports are made using the `ErsRep()` function. This function takes at least three arguments. The first is a flag argument which is explained later. The second is a standard modified status argument (note, it is the second argument, not the last). The third argument is a C `printf()` style formatting string. In the simplest case, this is the text to be output. Alternatively, you can use the C `printf()` formatting and specify the format arguments as extra arguments to the function call.

Unlike most other DRAMA functions, `ErsRep()` will operate with the status set bad. In addition, the value of status when the call is made will be associated with the error report and could be accessed by the user interface code responsible for outputting the message to the user (although no current user interfaces make use of this feature). The value of status will be unchanged unless an error occurs in `ErsRep()`. Application code which detects and error should set status bad and then report any context information using `ErsRep()`, for example—

Example   11.1   Calling   ErsRep

```
void myfunc(int bytes, char **p, StatusType *status)
{
    if (*status != STATUS__OK) return;
    ...

    if ((*p = malloc(bytes)) == 0)
```

```
    {
        *status = APP__MALLOCERR;
        ErsRep(0, status, "Error mallocing %d bytes", bytes);
    }
    ...
}
```

Multiple calls may be made to `ErsRep()`, resulting in a set of messages. This allows you to output multiple lines and add context in high level functions. For example—

Example 11.2 Add Context to messages

```
    ...
    myfunc(sizeof(name),name, status);
    if (*status != STATUS__OK)
    {
        ErsRep(0, status, "Couldn't get space for string");
    ...
```

The string supplied to `ErsRep()` should contain only printable characters. Any control characters should be striped out by the user interfaces, but just in case they are errant, it is better to ensure you don't supply them.

Ers Flags.

Various flags may be supplied to the `ErsRep()` call. These flags allow for formatting options and can provide hints to the user interface on how a message should be displayed.

Table 11.1 ERS Flags.

Flag	Meaning
ERS_M_NOFMT	Don't format the string. Any formatting arguments are ignored and the format string is used as specified.
ERS_M_HIGHLIGHT	Suggest to the user interface that the message should be highlighted.
ERS_M_BELL	Suggest to the user interface that the terminal bell (or an equivalent) should be rung when the message is output.
ERS_M_ALARM	Suggest to the user interface that the user should acknowledge this message.

Note that the user interface flags are suggestions only. Particular user interfaces may decide not to implement those functions.

Outputting error reports.

The `ErsRep()` call does not cause the messages to be output to the user. It saves them for later delivery. In DRAMA programs, this is done when your action reschedules or completes. This ensures that all associated messages are delivered at the same time and allows for the stacking control functions described below.

In some applications, it may be desirable to report an error but then continue. You can trigger delivery of all reported messages using the `ErsFlush()` function. It takes one argument, which is a modified status argument. This function will work with status bad and if delivery is successful, will clear status.

An additional function, `ErsOut()`, takes the same arguments as `ErsRep()`, but does an implicit `ErsFlush()` after reporting the message.

Control of message output.

The major reason for deferring the output of error messages is to allow final delivery of error messages to be controlled by applications software. Its use can be demonstrated by the following example[11].

Consider a subroutine, say "`helper()`", which detects an error during execution. The subroutine "`helper()`" reports the error that has occurred, giving as much contextual information about the error as it can. It also returns an error status value, enabling the software that called it to react to the failure appropriately. However, what may be considered an "error" at the level of the subroutine "`helper()`", eg. an "end-of-file" condition, may be considered by the calling module to be a case which can be handled without informing the user, eg. by simply terminating its input sequence. Although the subroutine "`helper()`" will always report the error condition, it is not always necessary for the associated error message to reach the user. The deferral of error reporting enables application programs to handle such error conditions internally.

Here is a schematic example of what "`helper()`" might look like—

Example 11.3 Function "helper"

```
void helper (char *line, StatusType *status)
{
    if (*status != STATUS__OK) return;
    ...
/*
 *   Check for end-of-file error
 */
    if (feof(instream))
    {
        *status = MYAPP__ENDFILE;
        ErsRep(ERS_M_NOFMT, status, "End of input file reached");
    }
/*
 *   Check for input error
 */
    else if (ferror(instream))
    {
        *status = MYAPP__INPUTERR;
        ErsRep(0, status,
```

---

[11]This example is largely and adaptation of one taken from the Starlink manual "EMS — Error Message Service Programmer's Manual", by Paul Rees. Starlink User Note 4.3.

```
                        "Error encounted during data input, errno = %x",
                        errno);
        }
        ...
}
```

Suppose "`helper()`" is called and reports an error, it returns with status set to a non-zero value. Now suppose that the subroutine "`helped()`" calls "`helper()`" and wishes to defer any messages from "`helper()`" so that it can decide how to handle the error conditions itself, rather then troubling the user with spurious messages. It can do this by calling the routine `ErsPush()` before it calls "`helper()`". This routine starts a new "error context", which is independent of any previous error messages. A return to the previous context can be made later by calling `ErsPop()`. The new context ensures that no existing error messages, waiting for delivery, will be lost though this mechanism. Calls to `ErsPush()` and `ErsPop()` should always occur in matching pairs and can be nested if required.

By calling `ErsPush()` before calling "`helper()`", the function "`helped()`" can now handle the error condition itself in one of three ways.

- By calling `ErsAnnul()`, which "annuls" the error, deleting any messages reported by "`helper()`" and resetting status to zero. This effectively causes the error to be ignored. For instance, it might be used if an "end-of-file" condition was expected, but was to be ignored and some appropriate action taken instead. (A call to `ErsRep()` could also be used after `ErsAnnul()` to replace the initial error condition with another more appropriate one, although this is not often done.)

- By calling `ErsFlush()`, which "flushes" the error, sending any deferred error messages in the current context to the user and resetting status to zero. This notifies the user that a problem has occurred but allows the application to continue anyway. For instance, it might be used if a series of files were being processed: If one of these files could not be accessed, then the user could be informed of this by calling `ErsFlush()` before going to process the next file.

- By doing nothing but restoring the previous error context by calling `ErsPop()` (which needs to be done in the above cases as well). Any messages in the context being removed are transferred to the previous context and will be reported as normal.

This example demonstrates the use of `ErsAnnul()`. It shows how an "end-of-file" condition from "`helper()`" might be detected and annulled.

Example 11.4 Function "helped"

```
void helped (StatusType *status)
{
    if (*status != STATUS__OK) return;
    ...
```

```
/*
 *  Create a new error context
 */
    ErsPush();
/*
 *  Any error messages from "helper()" are now deferred
 */
    helper(line, status);
/*
 *  Trap end-of-file and annul any reported messages
 */
    if (*status = MYAPP__ENDFILE)
    {
        ErsAnnul(status);
        eof = 1;
    }
/*
 *  Release the current error context
 */
    ErsPop()
}
```

sprintf( )

We take a slight side path at this point to examine a couple of routines provided by ERS which it needs itself but which solve a common problem.

Many C Run-time library routines which write to strings do not support any technique to determine the maximum length of the string which is being written. As a result, it is easy to overwrite the stack. One such routine, "sprintf()" is required by ERS. In order to avoid stack related problems, a special version of this routine and an associated routine, "vsprintf()", were written. The routines ErsSPrintf() and ErsVSPrintf() provide an extra argument over sprintf() and vsprintf(). The extra argument is before the other arguments and indicates the maximum length of the output string.

It is suggested you use these ERS routines instead of the C Run time library routines in any case where the maximum length of the resultant string can not be determined. This will result in fewer problems with stack overwrites.

Use in other applications.

The ERS library is not dependent on DRAMA itself and can be used in stand-alone applications and libraries. It is necessary through to consider how and when messages are output. For example, you would normally have to provide somewhere in your application, a routine to be invoked to perform output. If this is not provided, the default behaviour is to output error reports immediately to "stderr". See the ERS manual[vi] http://www.aao.gov.au/drama/doc/ps/ers_spec_4.ps)

Part 2 - Building systems

12.Control tasks — Getting Paths

Overview

A DRAMA control task is a task that controls other tasks. It is used to tie together a set of tasks, hiding complex interactions from the user. They allow the modular design of systems by allowing sets of simple tasks to be implemented to control logical parts of a system. These simple tasks are often associated with a particular logically stand-alone job, such as controlling a spectrograph, performing particular data processing etc. They can often be tested in a stand alone mode during testing and debugging. Control tasks control sets of these tasks to create more complex systems.[12]

In DRAMA, any task may be a control task. The author will just call a larger range of DRAMA routines to initiate messages to other tasks and handle the responses. This chapter introduces the concepts and routines required.

DRAMA Control   Concepts

In many systems which provide message sending, the sending functions block if a reply is expected. This can produce systems which are not responsive at particular times, particularly when things start to go wrong. DRAMA tries very hard to avoid blocking operations. At the basic levels, any DRAMA operation which sends a message just initiates the sending of the message. You are required to reschedule to await any response.

Higher level DRAMA procedures can hide this rescheduling, giving the impression that an action is blocking, whilst allowing the task to remain active. Below is an example using such routines.

Example   12.1   High level   message   sending

```
{
    DitsTaskParamType TaskParams;     /* Load parameters */
    SdsIdType arg;
    DitsPathInfoType info =
        { MESSAGEBYTES, MAXMESSAGES, REPLYBYTES, MAXREPLIES};
    DitsPathType path;
    ArgNew(&arg, status);
    ArgPuti(arg, "Argument1", 1234, status);
    strncpy(TaskParams.ProcName, "targetName",
```

---

[12]In the older Starlink ADAM systems, control tasks were special in both their concept and implementation,   you wrote a "C Task". This separation   does not exist in DRAMA.

```
                sizeof(TaskParams.ProcName));
    DulLoadW(targetTask, targetNode, 0, &info, targetProgram,
            0, 0, &TaskParams, TIMEOUT, &path, status);
    DulMessageW(DITS_MSG_OBEY,  path,  "HELLO",  arg,  TIMEOUT,
status);
}
```

This code loads a specified program if it is not already running and sends an Obey message to it. We will look at these higher level procedures after coming completely to terms with the basic concepts.

It is sufficient to say at this stage that for many simple cases, the higher level routines are more appropriate.

The first stage to understanding how to initiate messages and handle responses is to understand the two data types DRAMA uses to manage such operations.

Transaction id's

The first of these are the Transaction Id's. When you initiate a message to another task, the routine you invoke will return a transaction id. This "id" is of the integral type `DitsTransIdType`. Only two things are done with variables of this type, either they are passed to other DRAMA routines or they can be compared for equality or to zero. Zero is always an invalid transaction id.

The primary reason for the existence of transaction id's is to allow you to keep track of multiple concurrent messages. When you get a response to a message, you can get the transaction id associated with that message and work out what message is associated with. Actions which only handle one outstanding message at a time often ignore the transaction id.

You can associate an item of your choice with a transaction id. This item is of type "`void *`" so could be used to store almost anything. If you need to store more then one item with a single transaction, then malloc a structure, store the items in the structure and then associate the address of the structure with the transaction. A typical use of this associate is to allow a transaction specific routine to be invoked when responses are received. You use `DitsPutTransData()` to make the association and `DitsGetTransData()` to retrieve it.

Paths

To send messages to another task, you must have a communication channel to that task. DRAMA calls this a "Path" and its type is `DitsPathType`. As with transaction id's, paths are integral types which can be compared for equality or against zero, with zero always an invalid path. The setting up of a path involves the allocation of buffer space and other details required for sending and receiving messages. This involves messages being sent to and returned from the target task. As a result, this operation involves a transaction.

As with transaction id's, you can associate an item of your choice with a path. Use `DitsPutPathData()` to make the association and `DitsGetPathData()` to retrieve the data.

Setting up Paths.

The routine to initiate getting a path is `DitsPathGet()`. An older alternative with less flexibility is `DitsGetPath()`. The calling sequence for the former routine is as below.

Example 12.1 DitsPathGet

```
void DitsPathGet(
    char *name,
    char *node,
    int flags,
    DitsPathInfoType *info,
    DitsTransIdType *transid,
    DitsPathType *path,
    StatusType *status);
```

The "`name`" argument specifies the name of the task you want a path to. This name is that which was specified by the target task when `DitsAppInit()` was invoked. You can specify the machine on which the task is running using the "`node`" argument, or for convenience, you can also specify it as part of the "`name`" argument using the format "`taskname@nodename`". The "`node`" argument takes precedence.

In either case, the node name is only used if the task specified is not already known locally. A task is "known locally" if—

- The task is running on the same machine as the program calling `DitsPathGet()`.
- The task is on a different machine (not necessarily the one specified in the "`node`" argument) but another task on the machine where `DitsPathGet()` is being invoked has already got a path to a task of the same name.

Note, this means that a given DRAMA system can have only one task of a given name at the one time. A DRAMA system is defined as a set of communicating DRAMA tasks, normally being run by one user (the same user name) across one or more machines in a network. It is possible to have multiple tasks using the same name run by the same user on different machines, as long as no attempt is made to join the system by tasks on one machine trying to communicate with tasks being run by the same user on the other machine.

The "`flags`" argument allows various options to be specified being

`DITS_M_FLOW_CONTROL` The connection should be flow controlled. See the IMP documentation for more details about the effect of flow control. XXX should expand on this.

DITS_M_PG_IMMED   If set, then two cases which normally cause transactions to start won't. The first case occurs if we are already getting a path to the same task. In this case, if this flag is set, instead setting up a transaction to be triggered when the operation completes, we will return immediately with status DITS__FINDINGPATH. The second case is if we already have a path to this task. In that case, we normally set up a transaction to complete when the path is available for sending. But, if this flag is set, we return immediately with no transaction set up but status ok and `*transid` will be zero.

The "`info`" argument provides associated information to the `DitsPathGet()` routine, in particular, the buffer sizes, explained below. The "`path`" and "`transid`" arguments are also explained below.

Networking

In order to get a path to a remote task, you must have the DRAMA networking software running on both machines. These programs manage the communications across the networks, relieving individual DRAMA programs of the complexities often involved in network communications.

You start the networking tasks using the "`dits_netstart`" command, available on all platforms on which DRAMA runs. To shutdown the networking, use "`dits_netclose`".

Additionally, you should note that Unix systems do not provide proper exit handlers. So although DRAMA tries very hard to tidy up the resources it uses, it is not always able to. Semaphores, Shared memory, Message queues and mapped files may be left about if a DRAMA task is killed using "`kill -9`" or dies under some conditions such as "segmentation faults". To tidy up in such cases, use the "`cleanup`" command. This will tidy up all such resources (but does kill running tasks). See the "`-h`" option to "`cleanup`" for more information.

Buffer sizes and usage

When a path to another task is set up, buffer space is allocated for messages to be sent between the two tasks. Two buffers are involved, a buffer for sending messages to the target task and a second buffer for replies sent back by that task.

The former buffer is allocated in the target task, using space taken from its "Global Buffer Space". The size of this global buffer space is set when `DitsAppInit()` is invoked. Thus to get a path to a task, sufficient space must be available in the target tasks global buffer. The "`info`" argument to "`DitsPathGet()`" allows you to specify the size, using two values. Set the element "`MessageBytes`" to the number of bytes required for a typical message to the target task. Set the "`MaxMessages`" argument to

the number of such messages which may be sent in the maximum time required for the target to process one such message. Typically, values of 800 to 1000 are used for the first value and 1 to 10 for the second.

The second buffer is allocated in the task calling `DitsPathGet()`, using space taken from it's own "Global Buffer Space". Again there are two elements in the "`info`" argument, "`ReplyBytes`" and "`MaxReplies`", with similar uses. Again typical values for the former item are 800 to 1000. For the second value, 10 is more typical than lower values.

In both cases, if you are sending messages with large argument structures, such as images, you will need to increase the relevant values. The routine `DitsGetMsgLength()` can be used to determine the size of a DRAMA message.

Get Path results

After a call to `DitsPathGet()`, there are two or three possibilities (depending on the flags which have been set) being—

- Immediate failure. Status will be set bad. There are several possibilities, the most common being that no node name is supplied and the task is not known on the local machine, or that the task is remote and the network tasks are not running.

- You have to wait for completion. In this case, "*transid" is set to a non-zero value. "path" is set to the path to the task but cannot be used until the response is received. Normally this involves some real work, messages have to be sent to set up the path. But in the case where the path already exists, you are simply notified when the buffers to that path are empty, but see the next point for more information here.

- If you specified the flag `DITS_M_PG_IMMED`[13] then you may get a case where status is good but "*transid" set to zero. This indicates that a path is already available to be used and you can use it immediately

It is possible to specify a null pointer for the transid argument. In this case, you are doing an inquiry only. The "`info`" argument is ignored. If your task already has a path to the

---

[13]In older versions of DRAMA the behavior was as if this flag was always set and the flag did not exist. From DRAMA V1.0, this was changed. In addition, from DRAMA V1.0, you may get a path to the same task multiple times concurrently. Previously, you would get the status `DITS__FINDINGPATH` on subsequence attempts to get a path before the first succeeded. It is believed that all code which worked previously will still work, except for code which assumed the path would already be known and did not check for a non-zero value being return for the transaction id. Such code should be changed to specify zero (0) for the transaction id address. This will ensure no reschedule is required.

task, it is returned. If that path is useable immediately, status will be Ok. If it is not useable (probably indicates a Get Path operation is outstanding) then status is set to `DITS__INVPATH`. If the path does not exist, the status is set to `DITS__UNKNTASK` and path is set to 0.

Waiting for completion

If your action must wait the completion of the Get Path operation you must set up an action reschedule. You need to put a reschedule request, specify a new handler if desired and enable a timeout is required before returning to the fixed part.

Example 12.2 Handling DitsPathGet

```
DitsPathGet(task, node, 0, &buffers, &transid, &path, status);
if (*status == STATUS__OK)
{
    DitsPutRequest(DITS_REQ_MESSAGE, status);
    DitsPutObeyHandler(PathHandler, status);
    DitsPutDelay(&timeout, status);
```

Here the reschedule request is `DITS_REQ_MESSAGE`. This request indicates we are waiting for message.

Get Path handler The get path handler routine (put with `DitsPutObeyHandler()` above) is invoked when the system has successfully set up the path, a failure has occurred whilst setting up the path or the timeout (if specified) occurs.

You can find out the reason for the event using `DitsGetEntReason()`, which in this case will return one of—

`DITS_REA_PATHFOUND`This indicates the path was set up successfully and you can now send messages using the path variable returned by `DitsPathGet()`. You can used `DitsGetEntTransId()` to get the transaction id associated with the event, which had previously been returned by `DitsPathGet()`. You can also get the path associated with the event using `DitsGetEntPath()`.

`DITS_REA_PATHFAILED`   This indicates the system failed to set up a path to your task. Again you can use `DitsGetEntTransId()` to get the transaction id associated with the event. You can use `DitsGetEntStatus()` to get the reason for the failure.

`DITS_REA_RESCHED`   This event type indicates a timer based reschedule, normally used to implement timeouts. You should note that such events are not considered errors and `DitsGetEntStatus()` will return `STATUS__OK`. In addition, the Get Path event is still outstanding and may still occur. To disregard it invoke `DitsLosePath()` on the path involved.

Having handled this event, assuming you got the success message, you are now ready to send messages to the task in question.

13. Control tasks - Sending Messages

Overview

Once you have a path to a task (see chapter 12), you can start sending messages to it. Once again, we will examine the basic facilities provided by DRAMA. Remember that a later chapter will introduce the higher level routines which will make many task implementations simpler.

Message Sending Calls

The basic routine to send a message to DRAMA task is `DitsInitiateMessage()`.

Example 13.1 DitsInitiateMessage

```
void DitsInitiateMessage(
    int flags,
    DitsPathType path,
    DitsTransIdType *transid,
    DitsGsokMessageType *message,
    StatusType *status);
```

The flags argument specifies a number of options. "`path`" is a path returned by `DitsPathGet()`. If a message is successfully started, a DRAMA transaction is initiated and "transid" will contain the transaction id. The details of the message to be sent are set up in the "message" argument. These include the name of the message (i.e. the action name in an Obey message, the Parameter name in a parameter Get message), the message type and the SDS id of any argument SDS structure a copy of which is sent with the message. See the routine description for full details.

The above call is the most efficient way to send a DRAMA message. This is particularly true when the same message is to be sent multiple times as the "message" structure need only be set up once. Note that this argument must be write able. Although it is returned with the same value supplied on entry, it is temporarily modified during the implementation of this function.

DRAMA provides wrap around s' of `DitsInitiateMessage()` for its most common cases. These avoid the need for the caller to allocate and initialise the "message" structure argument. The routines provided are `DitsObey()`, `DitsKick()`, `DitsGetParam()` and `DitsSetParam()`. They have similar calling sequences, for example

Example 13.2 DitsObey

```
void DitsObey(
    DitsPathType path,
    const char name,
    SdsIdType arg,
    DitsTransIdType *transid,
    StatusType *status);
```

Here you specify the path, action name for the obey message and SDS structure id. A transaction id is returned on success.

Handling message replies

Replies to any messages started using the above calls are handled in the same way as with Get Path messages. You reschedule to await the replies and use `DitsGetEntReason()` and `DitsGetEntStatus()` to get details of the response. Normally, `DitsGetEntReason()` will return one of the following codes

`DITS_REA_COMPLETE` This indicates the subsidiary task has handled the message and if an action was involved, the action has completed. The status value returned by the subsidiary task's action handler routine in response to the message is available by calling `DitsGetEntStatus()`.

`DITS_REA_MESREJECTED`   This code indicates the target task has rejected the message. In the case of Obey messages this indicates the subsidiary task's user written action code was never invoked. Again, you can get the status associated with the error using `DitsGetEntStatus()`. Common causes of this response are there being no action of the given name or the action is not spawnable and is already active.

`DITS_REA_TRIGGER` This code indicates an intermediate message has been received from the subsidiary task. If associated with an action, the action is continuing. The subsidiary task can generate such messages using the routine `DitsTrigger()`.

`DITS_REA_DIED`   The subsidiary task has died. The actual meaning in DRAMA is that your task has lost its connection to the subsidiary task before handling of the message originated by your task completed. Again you can use `DitsGetEntStatus()` to get the details of the failure, which could indicate the task has died or just that we have lost all connections to the machine on which the task is running (possibly due to network failures).

As with Get Path messages, you can use `DitsGetEntTransId()` to get the transaction id associated with one of these messages, that is, the value which was returned by original `DitsInitiateMessage()` call. You can also use `DitsGetEntPath()` to get the path to the subsidiary task. An additional call, `DitsGetEntName()` allows you to retrieve the name associated with the original message (the action name for Obey messages).

Reply Argument structures

It is possible for subsidiary tasks to associate SDS structures with reply messages, the arguments to the reply. In the case of action handler routines, this would have been done by calling `DitsPutArgument()`. See chapter 7 for details. By doing this, any amount of information can be passed from the subsidiary task up to the originating task.

The SDS id of any such argument structure can be obtained using `DitsGetArgument()`, the same call which is used to obtain the argument associated with the original obey message. `DitsGetArgument()` will return the argument associated with the message which caused the entry, hence for the first entry to an action, it is the argument associated with the Obey message which starts the action. For subsequent entries, it is the argument associated with the message which triggered the entry.

If no such argument is available, `DitsGetArgument()` will return the null SDS id of 0 (zero). This SDS id is free-ed implicitly when your action reschedules or completes. Also, it refers to an external type SDS item. As a result of these features if you need to keep the SDS structure about after you have rescheduled, you should copy it, using say `SdsCopy()`.

Special Reply Handling

Some replies to messages originated by your task do not result in your action being rescheduled. They are handled transparently by the DRAMA fixed part (the Main Loop code). Normally these messages are intended for the user interface and DRAMA helps you out by forwarding them automatically to the originating user interface. These messages have the following reason codes associated with them—

`DITS_REA_MESSAGE`   These reply messages are similar to trigger messages in that the action in the subsidiary task has not completed. They are used to send informational messages to the user interface and are generated when the subsidiary task invoked `MsgOut()`.

`DITS_REA_ERROR`   As per `DITS_REA_MESSAGE`, but for messages generated using the Ers package.

It is possible to switch off the default handling of these replies using `DitsInterested()`. You might do this to implement higher level features similar to those implemented by the ERS package. In addition, you can use `DitsNotInterested()` to switch on default handling of these and other reply types. When handling these messages, the actual message for the user interface is contained in the SDS argument structure associated with the message. These have well defined format documented in the DITS Manual [vii] (see http://www.aao.gov.au/drama/doc/ps/dits_spec_5.ps).. The DUL library (XXX) reference) provides some help here. The various DUL library routines prefixed by `DulErs` implement such a scheme. After invoking `DulErsInit()`, which itself calls `DitsInterested()`, you invoke `DulErsMessage()` to handle each ERS message. You can then use `DulErsRep()` and `DulErsAnnul()` to control such messages. `DulErsFinished()` is used to revert to the normal mode, whilst `DulErsPutRequest()` replaces calls to `DitsPutRequest()`.

Simple control task example.
We implement below a simple control task using operations defined so far. Remember that the DUL routines provide simpler but less general implementation.

This program implements a "RUN" action in addition to the normal "EXIT" action. The "RUN" action calls `DitsPathGet()` to get a path to the task "DRAMAHELLO", or whatever task name was specified as an argument to the program. If the call to `DitsPathGet()` is successful, it reschedules to await completion, specifying the routine `PathWaitHandler()` as the routine to be invoked when the operation completes.

The `PathWaitHandler()` routine examines the reason it was entered using `DitsEntReason()`. If this indicates the path was found, it sets up an argument structure and obeys the action "HELLO" to the task using `DitsObey()`. It sets up a reschedule with the routine `ObeyHandler()` as the handler.

The `ObeyHandler()` routine also examines the reason it was entered, reporting any error.

Example 13.3 A Simple Control Task

```
#include "DitsTypes.h"          /* Basic Dits types */
#include "DitsSys.h"            /* Initialisation functions */
#include "DitsMsgOut.h"         /* MsgOut    */
#include "DitsFix.h"                /* For DitsPutRequest */
#include "sds.h"                /* For Sds stuff */
#include "arg.h"                /* For Arg routines */
#include "DitsUtil.h"           /* For DitsGetErrorText */
#include "DitsInteraction.h"    /* For communication stuff */


#define BUFSIZE 20000           /* Global buffer size */
#define TASKNAME "CTASK"        /* Name of this task */
#define TIMEOUT 10              /* Timeout for message operations */


static void Run(StatusType *status);/* Prototypes of handlers */
static void Exit(StatusType *status);


const char *targetTask = "DRAMAHELLO";
extern int main(int argc, char **argv)
{
    StatusType status = STATUS__OK;
/*
 *  Define the actions supported by this task.
 */
    static DitsActionDetailsType Actions[] =
        { { Run, 0, 0, 0, 0, 0, "RUN"},
          {Exit,  0, 0, 0, 0, 0, "EXIT"} };

/*
 *  The target task can be optionally specified on the
 *  command line.  This allows us to run this example across two
```

```
 *   machines.  Note, this is the ctask program command line,
 *   not the "ditscmd" command line.
 */
    if (argc >= 2)
        targetTask = argv[1];

/*
 *   All the normal stuff.
 */

    DitsAppInit(TASKNAME, BUFSIZE, 0, 0, &status);
    DitsPutActions(DitsNumber(Actions),Actions, &status);
    DitsMainLoop(&status);
    return (DitsStop(TASKNAME, &status));
}
/*
 *   Default message buffer sizes.
 *   See DitsAppInit and DitsPathGet routine descriptions.
 */
#define MESSAGEBYTES 400
#define MAXMESSAGES 5
#define REPLYBYTES 800
#define MAXREPLIES 12

DitsPathType HelloPath;          /* Path is stored here */


/*
 *   Function prototypes
 */
static void PathWaitHandler(StatusType *status);
static void ObeyHandler(StatusType *status);
/*
 *   Implements the RUN action, which gets a path to the
 *   targetTask and send the action DRAMAHELLO to it.
 */
static void Run(StatusType * const status)
{
    DitsTransIdType transid;
    DitsPathInfoType info = {   MESSAGEBYTES, MAXMESSAGES,
                                REPLYBYTES, MAXREPLIES};
    if (*status != STATUS__OK) return;
/*
 *   Try for the path.
 */
    DitsPathGet(targetTask, 0, 0, &info, &HelloPath,
        &transid, status);
    if (*status != STATUS__OK)
    {
/*
 *   Failure when getting path.  Report and return the bad status.
 */
        ErsRep(0, status, "Failed to get path to DRAMAHELLO");
    }
    else
    {
        DitsDeltaTimeType timeout;
        DitsDeltaTime(TIMEOUT, 0, &timeout);
/*
```

```
 *       Reschedule to wait for path.
 */
        DitsPutDelay(&timeout, status);
        DitsPutRequest(DITS_REQ_MESSAGE, status);
        DitsPutHandler(PathWaitHandler, status);
    }
}
/*
 *   Invoked when the path wait reschedule (set up above) occurs
 */
static void PathWaitHandler(StatusType *status)
{
    if (*status != STATUS__OK) return;
    switch (DitsGetEntReason())
    {
        case DITS_REA_RESCHED:
/*
 *          Timeout.  We invalidate the path return.  The
 *          Transaction will be orphaned automatically on
 *          action completion.
 */
            DitsLosePath(HelloPath, status);
            *status = DITS__APP_TIMEOUT;
            ErsRep(0, status,
                "Timeout trying to get path to DRAMAHELLO");
            break;
        case DITS_REA_PATHFOUND:
        {
/*
 *          Success, send the obey.
 */
            DitsTransIdType transid;
            DitsDeltaTimeType timeout;
            SdsIdType arg;
/*
 *          Create the argument to the action.
 */
            ArgNew(&arg, status);
            ArgPuti(arg, "Argument1", 1234, status);
/*
 *          Send the obey
 */
            DitsObey(HelloPath, "HELLO", arg, &transid,
            status);
/*
 *          Set up timeout and reschedule.
 */
            DitsDeltaTime(TIMEOUT, 0, &timeout);
            DitsPutDelay(&timeout, status);
            DitsPutRequest(DITS_REQ_MESSAGE, status);
            DitsPutHandler(ObeyHandler ,status);
            break;
        }
        case DITS_REA_PATHFAILED:
/*
 *          Failure.  Report it.
 */
            *status = DitsGetEntStatus();
```

```
                ErsRep(0, status,
                    "Failed to get path to task DRAMAHELLO:%s",
                    DitsErrorText(*status));
                break;
        default:
/*
 *          Something strange has happened.
 */
                DitsPrintReason(DitsGetEntReason(),
                    DitsGetEntStatus(),status);
                *status = DitsGetEntStatus();
                ErsRep(0, status,
        "Unexpected entry while getting path to DRAMAHELLO:%s",
                    DitsErrorText(*status));
                break;
    }
}
/*
 *  Invoked to handle completion of the obey message
 */
static void ObeyHandler(StatusType *status)
{
    if (*status != STATUS__OK) return;
    switch (DitsGetEntReason())
    {
        case DITS_REA_RESCHED:
/*
 *          Action has timed out
 */
                *status = DITS__APP_TIMEOUT;
                ErsRep(0, status,
                    "Timeout occurred waiting for results");
                break;
        case DITS_REA_COMPLETE:
/*
 *          Action had completed
 */
                *status = DitsGetEntStatus();
                if (*status != STATUS__OK)
                    ErsRep(0, status,
"Action %s to DRAMAHELLO completed with error status \"%s\"",
                        DitsGetEntName(),
                        DitsErrorText(*status));
                else
                    MsgOut(status,
                        "Action %s to DRAMAHELLO completed ok",
                        DitsGetEntName());
                break;
        case DITS_REA_MESREJECTED:
        case DITS_REA_DIED:
/*
 *          Failure sending message.
 */
                *status = DitsGetEntStatus();
                ErsRep(0, status,
    "Action %s to DRAMAHELLO failed with error status \"%s\"",
                    DitsGetEntName(),DitsErrorText(*status));
                break;
```

DRAMA   9 3

```
        default:
/*
 *          We don't expect any other entry, output an error
 *          message if one occurs.  Note that ERROR and
 *          INFORMATIONAL messages will be automatically
 *          forwarded by the fixed part.
 */
            DitsPrintReason(DitsGetEntReason(),
                DitsGetEntStatus(),
                status);
            break;
    }
}
static void Exit(StatusType * const status)
{
    DitsPutRequest(DITS_REQ_EXIT, status);
}
```

To run the above example, run one of the previous DRAMAHELLO examples, one where the program does not exit until it receives the EXIT action. Then run this example and use "ditscmd CTASK RUN" to initiate the Obey. The RUN action in the above task can be invoked any number of times.

The above techniques can be used for any of the DRAMA messages. In addition, remember that you can start multiple Get Path operations/messages in any stage of the RUN action, using the transaction id to distinguish there responses. See the "tea" and "coffee" example in the DRAMA DITS library source code directory. In particular, the files ctest.c, tea.c and coffee.c.

14.Loading Tasks, Handling Death

Overview

In any software system containing multiple tasks, it is probably desirable for one task to initiate the loading of the other tasks in the system. Loading tasks is a very system specific operation. DRAMA wraps up task loading operations in portable way whilst at the same time supporting the loading of tasks on different machines. Additionally, some time may pass between the program implementing a task being started and that task calling `DitsAppInit()`. During this period, a call to `DitsPathGet()` specifying the loaded task will fail. DRAMA helps here by providing notification to the loading task of the target task having called `DitsAppInit()`.

Related to task loading, is the handling the death of a task. This may occur intentionally or due to a hardware or software error. In order to allow the building of reliable systems, DRAMA tries very hard to ensure interested tasks are told when a task die. This chapter addresses both these issues.

In addition, the DRAMA task load facilities help in the cleaning up after tasks which have died. This feature is or particular interest where tasks have died uncleanly, leaving resources allocated. The DRAMA task loading facilities ensure such resources are deleted.

Task Loading

Generally, for task loading to work, the DRAMA networking tasks must be running. These are started using the command "`dits_netstart`". In addition to loading the network communication tasks, transmitter and receiver, this command loads a program known as the Imp Master task, which does the actual loading. To load a task on another node, you must have the DRAMA networking running on both machines involved.

Where all tasks to be loaded are running on the same machine as the loading task, the requirement to have the DRAMA networking running is sometimes annoying. To avoid it, you can specify the `DITS_M_MAY_LOAD` flag when you invoke `DitsAppInit()`. This allows a task to load another program itself.

To load another task, a DRAMA task calls `DitsLoad()`, which has the following format —

Example 14.1   DitsLoad

```
DitsLoad(
    char *Machine,        /* Machine (node) to load the task on*/
    char *TaskName,/* Name of program to load            */
    char *ArgString,     /* Program arguments                  */
    long int flags,/* Load flags                           */
    DitsTaskParamType *TaskParms,/* Load options        */
    DitsTransIdType *transid,  /* Transaction id           */
    StatusType *status)
```

In the above call, TaskName specifies the name of a program to run in a format which will be understood by the target machine.   Where the target machine is a VMS machine, this string is either an executable file specification or a CLI command name. File specifications can include logical names accessible to the invocation of the DRAMA networking running on the target machine. On VMS, loaded tasks are created by loading programs into a sub-process of the DRAMA networking Master task.

When the target is a Unix machine, the program is a file specification accessible to the invocation of the DRAMA networking. If prefixed with a slash "/" it is relative to the root directory.   Otherwise it is relative to the current directory of the DRAMA networking programs. Additionally, it may specify a program in the path of the DRAMA networking programs. Finally, environment variables may be used. If you specify "NAME:program" then "NAME" is considered to refer to an environment variable known to the DRAMA networking program. This environment variable is translated and prefixed to the rest of the string to get the program name. On UNIX, loaded tasks are created by loading programs into a process forked by the DRAMA networking Master task.

For VxWorks targets it is assumed that object modules implementing tasks have been loaded into memory. TaskName is a string which can be used to locate an routine entry point on the VxWorks machine. If the VxWorks symbol table is loaded, then this can be the routine name for the main entry point of the task. For systems without the VxWorks symbol table, other techniques are available, see the IMP manual for details.   On VxWorks, loaded tasks are created by creating a new VxWorks task and specifying the routine entry point as the start address.

The ArgString argument can be used to specify an argument string to be passed to the task to be run. This string is used to set up the "command line" arguments to be seen by the created program. You can specify NULL here if no arguments are required. See the `DitsLoad()` routine description for full details.

There are a number of flags which can be specified using the Flags argument. Most of the time, you won't need to set flags, but in some cases, things such as ambiguities in the way an operating system treats a name need to be considered. In addition, some flags indicate optional items in the TaskParams structure are set, such as the priority of the task to be loaded.

TaskParams is a structured item used to pass assorted information. This includes the name of the process into which the task is to be loaded (when the operating system supports process names), and optional information the existence of which is indicated by setting an appropriate flag. See the routine description for full details.

Task loading is a messaging operation— i.e. calling DitsLoad causes a message to be sent. As with all other DRAMA messaging operations, you need to reschedule to await the response. You will receive one or two responses, depending on the task being loaded and `DitsGetEntReason()` will return one of the following

DITS_REA_LOAD   The task has loaded successfully and has registered with
    DRAMA (IMP). This message is only received if the task being loaded is a
    DRAMA task. It is sent when the task calls `DitsAppInit()`. Once you receive
    this message, you can get a path to the task. The name the task has registered as
    (supplied to `DitsAppInit()`) is available to your task by calling
    `DitsGetEntName()`. You should always use this name when you call
    `DitsPathGet()`, rather than assume the task loaded under a particular name.

DITS_REA_LOADFAILED   The load operation failed. The point at which this is
    sent is somewhat system dependent but would normally indicate a failure to find
    the file or a lack of resources. `DitsGetEntStatus()` can be used to obtain
    the failure status. In addition, the routines `DitsLoadErrorText()` and
    `DitsLoadErrorStatus()` can be used to obtain more information about the
    failure.

DITS_REA_EXIT   The task has exited. Assuming the task was loaded
    successfully, this message indicates the program has now exited for whatever
    reason. If `DitsGetEntStatus()` returns a non-zero value, then the final exit
    status of the program indicates a normal completion.   Otherwise
    `DitsGetEntStatus()` will contain a bad status value and
    `DitsLoadErrorText()` and `DitsLoadErrorStatus()` can be used to
    obtain more information about the failure.

DITS_REA_MESREJECTED   Unlikely but not impossible, this reason code
    indicates a failure in the processing of the load message.
    `DitsGetEntStatus()` can be used to obtain the status associated with the
    failure.

Task Loading   Example

This example is a modification of the Simple control task in the previous chapter. This
time, if we can't find the task (`DitsPathGet()` gives an error), we attempt to load it
before trying again. This is a common approach — check if the task is running, if not,
load it.

Example 14.2 A Control Task with Loading

```
#include "DitsTypes.h"          /* Basic Dits types */
#include "DitsSys.h"            /* Initialisation functions */
#include "DitsMsgOut.h"         /* MsgOut   */
#include "DitsFix.h"            /* For DitsPutRequest */
#include "sds.h"                /* For Sds stuff */
#include "arg.h"                /* For Arg routines */
#include "DitsUtil.h"           /* For DitsGetErrorText */
#include "DitsInteraction.h"    /* For communication stuff */


#define BUFSIZE 20000           /* Global buffer size */
#define TASKNAME "CTASK"        /* Name of this task */
```

```
#define TIMEOUT 10              /* Timeout for message operations */

static void Run(StatusType *status);/* Prototypes of handlers */
static void Exit(StatusType *status);


char targetTask[100] = "DRAMAHELLO";  /* Default task name */
char *targetProgram = "./dramahello-7";/*Default program name */
char *targetNode = 0;           /* Default node to load on */
DitsDeltaTime timeout;
extern int main(int argc, char **argv)
{
    StatusType status = STATUS__OK;
/*
 *  Define the actions supported by this task.
 */
    static DitsActionDetailsType Actions[] =
        { { Run, 0, 0, 0, 0, 0, "RUN"},
          {Exit,  0, 0, 0, 0, 0, "EXIT"} };

/*
 *  The target task, program to load and node on which to load
 *  and find the program can be optionally specified on the
 *  command line.  Note, this is the ctask program command line,
 *  not the "ditscmd" command line.
 */
    if (argc >= 2)
        strncpy(targetTask, argv[1],sizeof(targetTask);
    if (argc >= 3)
        targetProgram = argv[2];
    if (argc >= 4)
        targetNode = argv[3];
/*
 *  All the normal stuff.
 */
    DitsAppInit(TASKNAME, BUFSIZE, 0, 0, &status);
    DitsPutActions(DitsNumber(Actions),Actions, &status);
/*
 *  Set up a timeout to be used throughout
 */
    DitsDeltaTime(TIMEOUT, 0, &timeout);

    DitsMainLoop(&status);
    return (DitsStop(TASKNAME, &status));
}
/*
 *  Default message buffer sizes.
 *  See DitsAppInit and DitsPathGet routine descriptions.
 */
#define MESSAGEBYTES 400
#define MAXMESSAGES 5
#define REPLYBYTES 800
#define MAXREPLIES 12

DitsPathType HelloPath;        /* Path is stored here */

/*
 *  Function prototypes
```

```
 */
static void GetPath(StatusType *status);
static void AttemptLoad(StatusType *status);
static void PathWaitHandler(StatusType *status);
static void ObeyHandler(StatusType *status);

static int HaveAttemptedLoad;

/*
 *   Implements the RUN action, which gets a path to the
 *   targetTask and send the action DRAMAHELLO to it.
 */
static void Run(StatusType * const status)
{
    if (*status != STATUS__OK) return;
    HaveAttemptedLoad = 0;
    GetPath(status);
}
static void GetPath(StatusType * const status);
{
    DitsTransIdType transid;
    DitsPathInfoType info = {   MESSAGEBYTES, MAXMESSAGES,
                                REPLYBYTES, MAXREPLIES};
    if (*status != STATUS__OK) return;
/*
 *   Try for the path.
 */
    DitsPathGet(targetTask, 0, 0, &info, &HelloPath, &transid,
            status);
    if (*status != STATUS__OK)
    {
/*
 *   Failure when getting path.  Attempt load if we have not
 *   already done so, otherwise, report and return the bad
 *   status.
 */
        if (HaveAttemptedLoad)
            ErsRep(0, status,
                "Failed to get path to DRAMAHELLO");
        else
        {
            ErsAnnul(status);
            AttemptLoad(status);
        }
    }
    else
    {
/*
 *     Reschedule to wait for path.
 */
        DitsPutDelay(&timeout, status);
        DitsPutRequest(DITS_REQ_MESSAGE, status);
        DitsPutHandler(PathWaitHandler, status);
    }
}
/*
 *   Invoked to attempt a load
 */
```

DRAMA   9 9

```
static void AttemptLoad(StatusType *status)
{
    DitsTaskParamType TaskParams;           /* Load parameters */
    DitsTransIdType transid;          /* Load transaction id */

    if (*status != STATUS__OK) return;
    HaveAttemptedLoad = 1;
    MsgOut(status, "Will attempt to load program %s",
            targetProgram);
    strncpy(TaskParams.ProcName, "targetName",
                sizeof(TaskParams.ProcName));
    DitsLoad(targetNode, targetProgram, 0, 0,
            &TaskParams, &transid, status);
/*
 *  Set up timeout and reschedule.
 */
    DitsPutDelay(&timeout, status);
    DitsPutRequest(DITS_REQ_MESSAGE, status);
    DitsPutHandler(LoadHandler, status);
}
/*
 * Invoked when a load message completes
 */
static void LoadHandler(StatusType *status)
{
    char *node = targetNode ? targetNode : "localhost";
                        /* Node name, for messages only */
    if (*status != STATUS__OK) return;
    switch (DitsGetEntReason())
    {
        case DITS_REA_LOAD:
/*
 *          Successful load.  Save the name the task registered
 *          as and get the path.
 */
            strncpy(targetTask, DitsGetEntName(),
                sizeof(targetTask));
            MsgOut(status,
        "Program \"%s\" loaded and registered on node %s as %s",
                targetProgram, node, targetTask);
            GetPath(status);
            break;
        case DITS_REA_LOADFAILED:
        case DITS_REA_MESREJECTED:
/*
 *          Load failed or message was rejected.  Report the
 *          failure
 */
            *status = DitsGetEntStatus();
            ErsRep(ERS_M_ALARM, status,
    "Program \"%s\" failed to load on node %s, status = %s.",
                targetProgram, node ,
                DitsErrorText(*status));
            if (DitsGetEntReason() == DITS_REA_LOADFAILED)
            {
                ErsRep(ERS_M_ALARM, status,
                "Associated system error is \"%s\"(%lx).",
                DitsLoadErrorText(),DitsLoadErrorStat());
```

100   14 Loading Tasks, Handling Death

```
                }
                break;
        case DITS_REA_EXIT:
/*
 *              Task has exited (before registering in this
 *              case, as a different handler is used if it
 *              registered successfully).  If there is no status
 *              associated with it, report application error,
 *              otherwise report the error.
 */
                if (DitsLoadErrorStat() == 0)
                {
                        *status = DITS__APP_ERROR;
                        ErsRep(ERS_M_HIGHLIGHT, status,
                "Program \"%s\" on node %s exited with status ok.",
                                targetProgram, node);
                }
                else
                {
                        *status = DitsGetEntStatus();
                        ErsRep(ERS_M_HIGHLIGHT, status,
                                "Program \"%s\", node %s exited",
                                targetProgram, node);
                        ErsOut(ERS_M_ALARM, status,
                "Associated system error is \"%s\"(%lx).",
                                DitsLoadErrorText(),
                                DitsLoadErrorStat());
                }
                break;
        case DITS_REA_RESCHED:
/*
 *              Timeout during load.
 */
                *status = DITS__APP_TIMEOUT;
                ErsRep(ERS_M_ALARM, status,
                        "Timeout loading program \"%s\" on node %s.",
                        targetProgram, node);
                break;
        default:
/*
 *              Should never happen.  We have accounted for all
 *              the expected reason codes.
 */
                *status = DITS__APP_ERROR;
                ErsRep(0, status,
                "Unexpected entry while loading DRAMAHELLO:%s",
                        DitsErrorText(*status));
                DitsPrintReason(DitsGetEntReason(),
                        DitsGetEntStatus(),
                        status);
                break;
    } /* switch */
}
/*
 *   Invoked when the path wait reschedule (set up by GetPath)
 *   occurs
 */
static void PathWaitHandler(StatusType *status)
```

```
{
    if (*status != STATUS__OK) return;
    switch (DitsGetEntReason())
    {
        case DITS_REA_RESCHED:
/*
 *          Timeout.  We invalidate the path return.  The
 *          Transaction will be orphaned automatically on
 *           action completion.
 */
            DitsLosePath(HelloPath, status);
            *status = DITS__APP_TIMEOUT;
            ErsRep(0, status,
                   "Timeout trying to get path to DRAMAHELLO");
            break;
        case DITS_REA_PATHFOUND:
        {
/*
 *          Success, send the obey.
 */
            DitsTransIdType transid;
            SdsIdType arg;
/*
 *          Create the argument to the action.
 */
            ArgNew(&arg, status);
            ArgPuti(arg, "Argument1", 1234, status);
/*
 *          Send the obey
 */
            DitsObey(HelloPath, "HELLO", arg, &transid, status);
/*
 *          Set up timeout and reschedule.
 */
            DitsPutDelay(&timeout, status);
            DitsPutRequest(DITS_REQ_MESSAGE, status);
            DitsPutHandler(ObeyHandler, status);
            break;
        }
        case DITS_REA_PATHFAILED:
/*
 *          Failure.  Attempt load if we have not already done
 *          so.  Otherwise, report the failure.
 */
            if *(HaveAttemptedLoad)
            {
                    *status = DitsGetEntStatus();
                    ErsRep(0, status,
                    "Failed to get path to task DRAMAHELLO:%s",
                        DitsErrorText(*status));
            }
            else
                    AttemptLoad(status);
            break;
        default:
/*
 *          Something strange has happened.
 */
```

```
                DitsPrintReason(DitsGetEntReason(),
                        DitsGetEntStatus(),
                        status);
                *status = DitsGetEntStatus();
                ErsRep(0, status,
        "Unexpected entry while getting path to DRAMAHELLO:%s",
                        DitsErrorText(*status));
                break;
    }
}
/*
 *   Invoked to handle completion of the obey message
 */
static void ObeyHandler(StatusType *status)
{
    if (*status != STATUS__OK) return;
    switch (DitsGetEntReason())
    {
        case DITS_REA_RESCHED:
/*
 *          Action has timed out
 */
                *status = DITS__APP_TIMEOUT;
                ErsRep(0, status,
                    "Timeout occurred waiting for results");
                break;
        case DITS_REA_COMPLETE:
/*
 *          Action had completed
 */
                *status = DitsGetEntStatus();
                if (*status != STATUS__OK)
                        ErsRep(0, status,
"Action %s to DRAMAHELLO completed with error status \"%s\""
                        DitsGetEntName(),DitsErrorText(*status));
                else
                        MsgOut(status,
                                "Action %s to DRAMAHELLO completed ok",
                                DitsGetEntName());
                break;
        case DITS_REA_MESREJECTED:
        case DITS_REA_DIED:
/*
 *          Failure sending message.
 */
                *status = DitsGetEntStatus();
                ErsRep(0, status,
    "Action %s to DRAMAHELLO failed with error status \"%s\"",
                        DitsGetEntName(),DitsErrorText(*status));
                break;
        default:
/*
 *          We don't expect any other entry, output an error
 *          message if one occurs.  Note that ERROR and
 *          INFORMATIONAL messages will be automatically
 *          forwarded by the fixed part.
 */
                DitsPrintReason(DitsGetEntReason(),
```

DRAMA   103

```
                          DitsGetEntStatus(),
                          status);
                break;
        }
}
static void Exit(StatusType * const status)
{
    DitsPutRequest(DITS_REQ_EXIT, status);
}
```

Task Death

DRAMA tries hard to notify your task if another task with which it is communicating dies. These notifications are triggered if the task dies for whatever reason or if communication with the machine running that task is lost. A special technique is required to ensure quick notification if the machine on which a task is running were to be subject to a hard reboot This is common with VxWorks machines and the technique is described later.

There are three notifications which may occur.

Subsidiary task death

A subsidiary task is one to which your task has sent a message. If the subsidiary task dies whist the transaction is outstanding, the transaction will complete immediately and your action will be invoked. `DitsGetEntReason()` will return `DITS_REA_DIED`. Both the previous control task examples handle this possibility. `DitsGetEntStatus()` will return the status associated with the loss of the task.

Loaded task death

If a task you have loaded dies, then the action which loaded the task will be rescheduled. DitsGetEntReason will return `DITS_REA_EXIT` or `DITS_REA_LOADFAILED`, depending on when the failure occurred. `DitsGetEntStatus()`, `DitsLoadErrorText()` and `DitsLoadErrorStat()` can be used to obtain more details of the failure.

Parent task death

Here we are talking about the death of a task which has invoked an action in your task, and where that action has not yet completed. The Kick handler of any such action will be invoked, if there is one. DitsGetEntReason will return `DITS_REA_DIED`. If there is no Kick handler, the parent task's death is ignored. In either case, it is no longer possible to get message through to the user interface in the normal way.

15.Control task tidbit's

Overview

The previous chapters have introduced the basic concepts and routines involved in building DRAMA control tasks. DRAMA provides an assortment of other routines which implement special features or wrap up the basic routines for the more common cases. This chapter looks at these facilities.

Action Blocking Techniques

Most complex DRAMA programs make extensive use of action staging to handle reception of incoming messages. This is by far the most efficient technique available to DRAMA. However, this approach tends to make simple programs more complicated. What is sometimes required is the ability for an action to "Block", whilst waiting for messages. This will allow a simple serial flow of control though action. At the beginning of chapter 12, you were shown high level interfaces using such techniques to simplify control tasks. Note here we are talking about the action blocking, not the task. Your task can continue to process messages and actions.

The generalised technique here is for your action to initiate all the transactions it wishes and then for the action to block to await the responses. The action is unblocked when any messages arrive and must loop through the event code if more then one such message is expected.

The call to use is `DitsActionWait()`. When you invoke this routine, it blocks the invoking action until a message is received for it. When such an event occurs, the action is unblocked and you can process the message in the normal way. The use of this routine has two significant effects on your task. First, this routine is somewhat less efficient then staging. This should be considered in programs which have to be very responsive. The inefficiency is due to this approach being dramatically different to the normal DRAMA approach to message handling. The second effect of concern is that if you have multiple actions blocking, they will be unblocked in the reverse of the blocking order, regardless of which action has message waiting. Thus, one action can lock another, you cannot get around this problem, even by kicking the earlier action or invoking `DitsKill()` on it.

Regardless of the problems mentioned above, this feature is useful, particularly for the simple case of sending one message and waiting for the response. It has been used as the basis of higher levels routines which make the control task examples in the previous chapters much shorter. See the following example.

Example 15.1 A Control task with Action Blocking

```
#include "DitsTypes.h"          /* Basic Dits types */
#include "DitsSys.h"            /* Initialisation functions */
#include "DitsMsgOut.h"         /* MsgOut    */
#include "DitsFix.h"            /* For DitsPutRequest */
#include "sds.h"                /* For Sds stuff */
```

```
#include "arg.h"                    /* For Arg routines */
#include "DitsUtil.h"               /* For DitsGetErrorText */
#include "DitsInteraction.h"    /* For communication stuff */
#include "dul.h"                    /* For Dul library       */

#define BUFSIZE 20000              /* Global buffer size */
#define TASKNAME "CTASK"           /* Name of this task */
#define TIMEOUT 10                 /* Timeout for operations */

static void Run(StatusType *status); /* Prototypes of handlers */
static void Exit(StatusType *status);

char targetTask[100] = "DRAMAHELLO";
char *targetProgram="./dramahello-7";
char *targetNode=0;
DitsDeltaTimeType timeout;

extern int main(int argc, char **argv)
{
    StatusType status = STATUS__OK;
/*
 *  Define the actions supported by this task.
 */
    static DitsActionDetailsType Actions[] =
        {       { Run, 0, 0, 0, 0, 0, "RUN"},
                { Exit,  0, 0, 0, 0, 0, "EXIT"} };

    if (argc >= 2)
        strncpy(targetTask, argv[1],sizeof(targetTask));
    if (argc >= 3)
        targetProgram = argv[2];
    if (argc >= 4)
        targetNode = argv[3];

    DitsAppInit(TASKNAME, BUFSIZE, 0, 0, &status);
    DitsPutActions(DitsNumber(Actions),Actions, &status);
    DitsMainLoop(&status);
    return (DitsStop(TASKNAME,&status));
}
/*
 * Default message buffer sizes.  See DitsAppInit and
 * DitsPathGet
 */
#define  MESSAGEBYTES 400
#define  MAXMESSAGES 5
#define  REPLYBYTES 800
#define  MAXREPLIES 12

static void Run(StatusType * const status)
{
    DitsTaskParamType TaskParams;     /* Load parameters */
    int flags = 0;                    /* Load flags */
    SdsIdType arg;
    DitsPathInfoType info =
                { MESSAGEBYTES, MAXMESSAGES,
                  REPLYBYTES, MAXREPLIES };
    DitsPathType path;
```

```
      ArgNew(&arg, status);
      ArgPuti(arg, "Argument1", 1234, status);

      strncpy(TaskParams.ProcName, "targetName",
          sizeof(TaskParams.ProcName));

      DulLoadW(targetTask, targetNode, 0, &info, targetProgram,
              0, flags, &TaskParams, TIMEOUT, &path, status);
      DulMessageW(DITS_MSG_OBEY, path, "HELLO", arg,
              TIMEOUT, status);
}

static void Exit(StatusType * const status)
{
    DitsPutRequest(DITS_REQ_EXIT, status);
}
```

The above example performs exactly the same function as the example 14.2. It attempts to get a path to a program and if that fails, it loads the program before trying again to get a path. It then sends an obey message to the task.

The routine `DulLoadW()` takes a combination of the arguments to `DitsPathGet()` and `DitsLoad()`. It calls both of these routines, using `DitsActionWait()` to wait for the responses. The `DuiMessageW()` sends a message and waits for the completion of the transaction before returning. In both cases you can supply a timeout as a floating point value.

Orphaned   Transactions

We have seen several ways for a DRAMA program to send messages to other tasks, be they OBEY/ KICK /GET /SET /MONITOR/ CONTROL messages, Get Path messages or Task Load messages. In all cases, the action which sent the message is considered the parent action and the message initiated is a transaction within another task. The message sending routines normally return a transaction id, which allows you to relate responses to your message (reschedule events) with the original message. As a result, you can have any number of transactions outstanding from one action.

A major implication of this is that normally actions which initiate messages continue to reschedule until they have handled all outstanding responses. But sometimes this is impractical. In most of the previous control task examples, timeouts were implemented throughout. If a response did not occur in the specified time, the action would be rescheduled anyway and would report an error before exiting. Of course the outstanding message could arrive at any time, or never. Since the parent action no longer exists, there is no clear place to deliver the response. DRAMA calls transactions whose parent action has completed "Orphaned transactions".

Handling orphaned transactions

DRAMA provides various facilities to handle events related to orphaned transactions. The default behaviour is for DRAMA to print a message on "`stderr`". In the simpler tasks, this is all you will want as transactions are normally only orphaned when things go wrong.

In a complex system, you may want to provide more appropriate handling of orphaned transactions. You will probably want to catch errors and output appropriate messages.

The first such approach is to provide a orphan handler routine. This is done by invoking `DitsPutOrphanHandler()`. This routine takes only two arguments, a routine of type `DitsActionRoutineType` and a status variable. It returns the previous orphan handler routine. The supplied routine will be invoked as a user interface (`UFACE`) context event handler routine. `UFACE` context is described in more detail in a later chapter.

The second approach to orphans is for a specify an action in your program to become the parent for all orphan transactions (to "adopt" the orphans). This is done by having the selected action invoke `DitsTakeOrphans()`. The first argument to this routine is an enumerated type with the following possible values

`DITS_TAKE_CURRENT`   The action will take over open of all currently known orphan transactions. It does not take over any transactions made orphans after this call.

`DITS_TAKE_FUTURE`   The action will become the parent of any transactions made orphans from the time if the call onwards.

`DITS_TAKE_BOTH`   The action takes over ownership of all currently orphaned transactions as well as any transactions made orphan from the time of the call onwards.

Once an action takes over ownership of orphan transactions, any events relating to such transactions are delivered to the action as if the transactions were it's own subsidiary actions. The only way you can tell the difference between an action which was orphaned and one which has not been orphaned (a transaction which was actually started by the action handling the orphans) is to use the call `DitsIsOrphan()`. This routine takes the transaction id returned by `DitsGetEntTransId()` and returns true if the transaction was "adopted". One important point to remember here, the orphan action must remain active, i.e., it must reschedule to await orphan transactions. If it completes, the adoption is forgotten.

Creating orphans on purpose

There are some cases where you have started a transaction but are not interested in the results. For example, when loading tasks using `DitsLoad()` there are often two messages expected — the one which indicates a task has loaded and the one which occurs when a task has died. You will normally want to wait for the first such message but may not be interested in the second. One thing you could do is just cause the action which invoked `DitsLoad()` to complete, which will make the transaction an orphan.

But, in some cases, you will want your action to continue, but not receive any more events from the `DitsLoad()` operation. You disown the transaction by invoked `DitsForget()`.

.

Buffer Full handling

A task receiving messages may not be able to handle messages as fast as the send can generate them. Eventually, the sender will find the buffer it is writing to full. This may occur for both local and remote receiving tasks. For the former, the case is more complex due to the involvement of the network. The slow point may either be the network or the receiving task.

There are a number of approaches to solving this problem. First, you could increase the size of the buffer you are sending too. This works in many cases but is a bit ad-hoc and won't work forever (if the sender never lets up and the receiver can run fast enough, you will always run out of buffer space.)

Alternatively, you can slow down the sending task. This works but is often not desirable as the receiver speed may depend on various factors, machine loading etc.

In addition, you could make the sending task retry after a failure to send a message, possibly with delays between each attempt. But this is a "dirty" approach, effectively polling for buffer space.

The best solution is for the sending task to be able to arrange to be notified when the receiver's buffers are empty. It can then send its message. DRAMA supports this approach using the `DitsRequestNotify()` routine. You should call this routine if you receive the `DITS__NOSPACE` status code when attempting to send a message (from `DitsObey()` etc). You specify the path you were trying to send the message along. You then return to await a reschedule event, where `DitsGetEntReason()` will return `DITS_REA_NOTIFY`.

The "Notification" technique is used automatically for action completion and error messages, ensuring that such messages are always sent, even though they may arrive late.

When remote tasks are involved, you must specify if the notification technique is to be used when a path is set up. This is necessary as network connections using notification are less efficient than connections without notification. A flag to `DitsPathGet()`

allows control over paths allocated by the sending task. Flags to `DitsAppInit()` effect connections to this task and the default for the older `DitsGetPath()` routine. The default flags to `DitsAppInit()` make connections to the task in question flow controlled but connections made by `DitsGetPath()` not flow controlled.

You cannot send a message on a connection for which a notification message has been requested. Again, for action completion and error messages, this is handled transparently. For other cases, you will an get error code, being on of

XXX

Two routines may help you determine appropriate buffer sizes for you tasks. The routine `DitsGetPathSize()` returns the maximum number of bytes which may be written to a path, assuming it is empty. The routine `DitsGetMsgLength()` indicates the size of a given message. Here you supply SDS id of the argument structure you intend attaching to the message. You must also indicate if the message will be a reply message (sent using `DitsTrigger()` or `DitsPutArgument()`) or a message initiated by this task (`DitsObey()` etc.).

Creating non-blocking programs

Tasks will behave better if they are always responsive to user commands. For example, having started a CCD exposure, the user may want to abort it or stop it at any time and expects things to work without hangs and pauses. In the standard DRAMA approaches, achieving this requires that time consuming activities are broken up using the rescheduling technique. Unfortunately such a style is not always possible or is sometimes very inefficient.

If you don't have the source code for a time consuming action, then DRAMA is limited in what it can do to help. The standard approach in this case would be to put the call to the time consuming routine in a separate task which only does that job for you. The parent task creates this task when necessary and can delete it using `DitsDelete()` if required to abort the operation.

Queue Peeking

In cases where you do have the source, DRAMA can help in a couple of ways. DRAMA allows you to "peek" at the messages in the queue and act accordingly. In the simplest case, you may desire that your program only reschedule when there is actually a message waiting for you. The routine `DitsMsgAvail()` can be used to determine this. It returns true if there are any messages waiting for the task.

Alternatively, you may only be interested in some particular messages, say a kick of your own action. Here you can use the routine `DitsPeek()`. This routine allows you to look for different messages in the message queue. It can handle some types of message immediately. Flags to this routine provide a lot of flexibility, so I refer you to the documentation of the routine.

Getting and Setting Parameters

Parameter values can be fetched using the `DitsGetParam()` routine. This has a similar interface to the `DitsObey()` routine and the value of the parameter is returned in the argument structure associated with the reply message. This SDS structure is of a type which can be accessed using the ARG routines, with the name of the relevant value being the name of the parameter.

Similarly, you can use `DitsSetParam()` to set the value of a parameter. You supply an argument structure which contains the new value of the parameter. It is up to the parameter system of the target task to determine if the value is acceptable. The reply message is just used to indicate success/failure, with no value being returned.

Multiple Parameter Get

As a convenience, you can get the values of multiple parameters with a single messages. XXX What is the interface XXX

The value returned with the success messages is again compatible with the ARG library, with in this case, a named entry for each parameter fetched.

Long parameter names

Normally, parameter names are limited to the same length as action names, being `DITS_C_NAMELEN` (20) characters including a terminating null. However, in some parameter systems it is desirable to allow longer parameter names. For example, in the SDP parameter system, longer names are used to support getting and setting the values of parts of a structured parameter. DRAMA provides a protocol to allow parameter systems to support long parameter names. The DRAMA routines `DitsGetParam()` and `DitsSetParam()` support long names transparently but other routines may not, in particular, any which uses the structure type `DitsNameType` to pass the action name to the routine. As a result, you need to know the protocol.

For messages to get the value of a parameter, the approach is to use the Multiple Parameter Get protocol, but to specify the long name. You don't have to specify multiple parameters to use this protocol and any parameter may have a long name.

For setting parameters, things are a bit more complicated. XXX How XXX

Task types and descriptions

You can give a task a *type* and a *description. .* A task type is potentially useful in situations where there may be multiple tasks doing a similar job and you are interested in sending messages to all such tasks which are registered or any task of the same type. The task type is just an integer item. DRAMA does not define any values, it is up to system designers to do so. Types also allows the use of a "Generic" interface, without having to rely on task names to the task type. For example, you may define a task type for a "Telescope Autoguider". There may be several different Autoguiders, depending on the

instrument being used. To avoid define that all such tasks should have the same name, you could just define a type and look for a task of that type.

Whist a task type is an integer value, enabling easy comparison, a task description is a character string. You can use this string for any purpose you desire, for example, to allow a utility to list details on all task present on a system.

You can set the type and description of a task using the function `DitsSetDetails()`. The default type for a task is zero.

Given a task's name, you can get it's type using `DitsGetTaskType()`. Likewise, you can use `DitsGetTaskDescr()` to fetch the description of a task.

Finding tasks.

Here I am referring to finding a task such that you can get a path to that task. For this to work, the task in question must already be known on the machine your task is running. i.e., it must be running on the same machine or if it is remote, a task on the machine must have already got a path to the target task[14].

The simplest case is where you want to find the name the first task of a given type. This can be one using `DitsFindTaskByType()`, which takes the type of the task you are looking for and returns the name of the first task of that type, if one of that type was found. Note that failure to find the task is not considered an error, a flag argument will indicate if a task was found.

A more complete interface is provided by `DitsScanTasks()`. This routine allows you to loop scanning the details of all tasks known to the machine.   Each call to the routine returns the details of the next task. Details returned include the task name, task type and task description. See the routine description for more details.

The real tidbit's

---

[14]A future version of DRAMA may remove the restriction.

16.Handling large data transfers

Overview

Chapter not written yet..

Example   16.1   TBD

```
/ *
```

17. The C++ Interface

Overview

Those writing DRAMA programs in C may wonder if DRAMA would have been better implemented in C++. DRAMA has objects (tasks and parameters), messages sent to those objects, and to some extent, the ability to implement inheritance (See the GIT library). Unfortunately, DRAMA is intended to be very portable and the C++ ANSI Standard was no where near complete, much less supported by compiler vendors, when work on DRAMA was commenced. After much thought, we determined that C was the only language which addressed all the architectures we needed to support in a consistent enough way.

That having been said, there is nothing to stop one using C++ to implement a DRAMA program. That DRAMA program will not be as portable as DRAMA itself, but that may not matter. The DRAMA C API is completely accessible from C++, largely because the header files are compatible. Here you are using the DRAMA C API as you would any other C API within a C++ program.

To help implement such programs, DRAMA provides various C++ classes which provide better implementations of some DRAMA features. These classes are just wrap arounds of the DRAMA C API, but having chosen to use C++, these classes provide a better interface.

Some of the standard libraries provide C++ wrap arounds. The SDS, ARG, SDP and GIT libraries. In addition, the DCPP library provides an interface to some of the DITS level DRAMA features. These interfaces are implemented to use one of the most common sub-sets of C++. The provision of Exceptions, Namespaces, Templates, Run Time type identification and the `bool` type by the compiler are NOT required. In some cases, DRAMA may make use of these features if available.

It must be noted the DRAMA's C++ interfaces are under active development and some problems have been noted with them (conceptual problems, not bugs). There will probably be an entirely new C++ interface at some stage which will be easier and much more Object-Oriented in approach. The work in this area is currently being done in JAVA and will be converted to C++ at some stage. As part of these changes, and alternative to the Dcpp package will be produced.

SDS and Arg C++ Interfaces

The class **SdsId** provides a C++ interface to SDS. Each C++ `SdsId` type variable represent an SDS ID (C type `SdsIdType`), not a complete SDS structure to which many SDS ID's may point. Each operation in the SDS C library which allocates an SDS ID maps to a constructor in the `SdsId` class (this mapping is not one-to-one). An

additional constructor is provided to import an existing `SdsIdType` into an `SdsId` variable. Member functions are provided for all the standard SDS operations.

The major benefit of the `SdsId` class is the use of the destructor to tidy up. When using `SdsId`'s, you don't normally have to worry
about deleting SDS items and freeing SDS ID's, as this is done automatically when the variable goes out of scope. Consider the following bit of C code, which will read an Sds item from a file, find the substructure named "`fieldData`" and list it using `SdsList()`. It goes to some trouble to ensure it tidies up correctly, even if a failure occurs part way through.

Example   17.1   SDS   Error   handling   in   C

```
/* List the item "fieldData" within the specified SDS file */
extern void CStyle(const char *filename, StatusType *status)
{
/*  Read an Sds item from filename */
    SdsIdType fileId;
    SdsRead(filename,&fileId,status);
    if (*status == STATUS__OK)
    {
        /* Read was ok, find fieldData. */
        SdsIdType fieldId;
        StatusType ignore = STATUS__OK;
        SdsFind(fileId,"fieldData",&fieldId,status);
        if (*status == STATUS__OK)
        {
            StatusType ignore = STATUS__OK;
            /* Find ok, list the item and tidy up. */
            SdsList(fieldId,status);
            SdsFreeId(fieldId,&ignore);
        }
        /* Tidy up from read */
        ignore = STATUS__OK;
        SdsReadFree(fileId,&ignore);
        SdsFreeId(fileId,&ignore);
    }
}
```

The same function written using the `SdsId` class is-

Example   17.2   SDS   Error   handling   in   C++

```
/* List the item "fieldData" within the specified SDS file */
extern void CppStyle(const char *filename, StatusType *status)
{
    /* This constructor reads an Sds item from a file */
    SdsId fileId(filename,status);
    /* This constructor finds the item fieldData */
    SdsId fieldId(fileId,"fieldData",status);
    /* List the field item */
    fieldId.List(status);
}
```

The `SdsRead()` call occurs within the first constructor, while the `SdsFind()` operation occurs in the second. In both cases, flags kept in the `SdsId` variable indicate

how the item was allocated, allowing the destructors invoked implicitly at the end of the routine to clean up. When routines which must navigate large structures are considered, the benefits are even more dramatic.

Inherited Status with constructors

You will notice in the above examples, that all the constructors include an inherited status argument. This is an unusual approach as in C++, it is normally assumed that a constructor either works or an exception is thrown. As mentioned above, the DRAMA C++ implementation does not assume exception support, which is not available under all C++ compilers. But, all the underlying C API calls have an inherited status argument. To support this, the constructors and destructors make use of a null SDS ID. If status is bad on entry or is set bad during the constructor, the type is still constructed, but with the null SDS ID. When the destructor is invoked, it sees the null SDS ID and does nothing.

Assignment and Copying

Assignment and copying of variables of this type has been prohibited by making the corresponding operators private to the class. This is done because

- It is not always clear what the user may want done (a deep or shallow copy for example)
- Handling of errors requires a status variable which is not available in these operations.

One result of this is that you cannot pass variables of this type to subroutines by value, you must pass them by either pointer or reference.

The member functions `ShallowCopy()` and `DeepCopy()` provide explicit control of copy and assignment operations and provide the required Status argument in the case of `DeepCopy()`.

In the case of the `ShallowCopy()`, in which the actual Sds Id is copied, you must indicate if the copy variable is to outlive the source. This is necessary since if it is to do so then the copy must be responsible for any required deletion and freeing operations, not the original variable. The function will make the necessary modifications to both variables to ensure the destructors work as required.

The `DeepCopy()` function uses the C function `SdsCopy()` to create a new structure which is a copy of the source structure.

In both cases, a destructor is automatically run on the Target `SdsId` variable before the copy is made. Alternatively, you can make use of a constructor which makes a deep copy of its argument to construct a new item.

Importing from SdsIdType variables

Normally, the constructor for an `SdsId` item can work out what should be done with an item when the destructor is invoked. But when `SdsIdType` variables are imported into an `SdsId` type, this is not possible. A special construstor is required.

The following constructor is used to create `SdsId` variables using an Sds id taken from an `SdsIdType` variable.

```
SdsId(SdsIdType id, bool free, bool del, bool readfree);}
```

You must use the three flags to tell `SdsId` how to handle this id in it's destructor. If `free` is set true, the id is free-ed by the destructor using `SdsFreeId()`. If `del` is true, then the item will be deleted using `SdsDelete()` or `SdsReadFree()`. The `readfree` flag is set true to indicate that `SdsReadFree()` should be used instead of `SdsDelete()`. All these flags have default values of false, telling the destructor to do nothing.

The following bit of code uses a call to `GitArgGetStruct()` to return an SDS id. We then setup an `SdsId` item to access it. Since the id returned by `GitArgGetStruct()` must be free-ed, but not deleted by the user when he is finished with it, we set the `free` `flag` to `true` and leave the rest of the flags at their default `false` values.

Example 17.3 Importing SdsIdType variables

```
SdsIdType id = 0;
char detailsName[30];
GitArgGetStruct(DitsGetArgument(), "XyDetails", 2, 0,
            sizeof(detailsName), detailsName, &id, status);
SdsId detailsId(id,true);
```

An alternative way to import a SDS id represented by a `SdsIdType` variable into an `SdsId` variable is the use of versions of the `ShallowCopy()` and `DeepCopy()` member functions which take an `SdsIdType` source instead of an `SdsId` source. The `DeepCopy()` function is otherwise equivalent to the standard case of an `SdsId` source. The `ShallowCopy()` function is similar to the constructor mentioned above in that you must specify flags which indicate what should happen to the ID when the variable goes out of scope.

Exporting to SdsIdType variables

In order to export the SDS ID represented by an `SdsId` variable to an `SdsIdType` variable, use the member function `COut()`.

```
COut(bool outlives, SdsIdType *id, bool *free,
      bool *del, bool *readfree);}
```

This function returns the appropriate `SdsIdType` in the `id` variable. You must indicate if the `SdsIdType` variable will outlive the `SdsId` variable by setting the `outlives` flag `true`. This will ensure that the destructor for the `SdsId` variable does not free the

ID. You can make use of the other arguments to determine what you should do with the ID in the case of it outliving the source. The flag pointers are optional and if set to the default of 0, are ignored, but otherwise indicate what the destructor would have done.

An alternative to `COut()` in the situation where it is clear the source variable will outlive the target is the conversion operator which will return an `SdsIdType`. This operator allows `SdsId` variables to passed to any function which expects an `SdsIdType`.

Static and Global variables

It is normally not appropriate to construct complex items, such as SDS structures, in the constructors of static and global items. In such cases, the default constructor allows you to define an `SdsId` which represents a null Sds ID. Your initialisation code can then construct the item into a local variable and use `ShallowCopy()` to copy it into the global. For example

Example 17.4 Shallow copy into Static SDS id

```
static SdsId paramId;/* Static item using default constructor */

/* Create the item an copy it to the static item*/

SdsId top("TOP",SDS_STRUCT,status);
paramId.ShallowCopy(top,true);        /* paramId outlives top */
```

You can check if a object is initialised using the boolean operator. It returns true if the object is initialsied.

Arg

The `Arg` class is derived from `SdsId`. Only two constructors are available. One of these is identical to the `SdsId` import constructor described above (This also implements the default constructor in both cases, using argument defaults).

The second constructor is used to create a new argument structure. Its first argument is a dummy logical argument, which can be given any value. Its purpose is to distinguish this constructor from the default constructor. The second argument is a status argument. The third argument is optional and allows you to specify an alternative name for the top level structure. The default is ``ArgStructure'', as normally used by the ArgNew(3)} function.

Member functions allow you to create a new structure using an existing item, write an item to a string and put and get the value of an item.   The following code creates a new `Arg` variable and puts the value "TRUE" into it as a character string

Example 17.5 Using Arg C++, creating structures

```
Arg arg(true,status);
arg.Put("Argument1","TRUE",status);
```

Remember that all the standard member functions of the `SdsId` class are also available.

Sdp

The `Sdp` class is a simple interface to the Simple DITS Parameter system (see the DITS manual[viii] - http://www.aao.gov.au/drama/doc/ps/dits_spec_5.ps). There are no constructors, just static member functions which make use of overloading to select the appropriate interface to Sdp. The implicit object you are operating on is the parameter system.

You should use calls like these-

Example 17.6 Using Sdp C++

```
static long c;
Sdp::Put("FRED",1,status);
Sdp::Get("FRED",&c,status);
```

Git

A number of classes are defined in the Git include file. The base type `Git` provides a simpler way of specifying the various flags used by the equivalent of the `GitArgGet*` functions (\cite{GIT_SPEC}).

GitBool

This class implements a boolean type which includes a member function to fetch values from an SDS structure using `GitArgGetL()`. This class provides a simpler interface to fetching boolean values from string or integer values in SDS structures.

The following code shows the declaration of a `GitBool` variable and the fetching of the value from an SDS item. (Note that
since a conversion exists for `SdsIdType` to `SdsId` we can use
`DitsGetArgument()` directly to get the SDS id).

Example 17.7 Using GitBool

```
GitBool Flag;
Flag.Get(DitsGetArgument(), "CONFIRM",1,status);
if (Flag)      /* Make use of conversion to bool operator */
    ...
```

Alternately, the constructor can do the Get operation immediately, allowing the above code to be replaced by

Example 17.8 Using GitBool constructor

```
GitBool Flag(DitsGetArgument(), "CONFIRM",1,status);
if (Flag)
    ...
```

In both the above cases, the strings accepted as True and False are as specified by `GitBool` (just "TRUE" and "FALSE" themselves). The following example shows the definition and usage of a class
(`ParkBool`) which inherits `GitBool` but provides alternative True and False value strings (PFA/ZENITH).

Example 17.8 GitBool alternative   strings

```
class ParkBool : public GitBool {
    private:
        static const GitLogStrType lookupTable[];
        const GitLogStrType * Lookup() { return lookupTable;};
    public:
    ParkBool(const SdsId & Id, const char * const Name,
             const int Position, StatusType * const status,
             const int Flags =  Git::Upper|Git::Abbrev))
        //Use Get not constructor, see C++ Ref manual r.12.7
          : Get(Id,Name,Position,status,false,Flags){ }
};
/* Define the static item defined in ParkPool */
const GitLogStrType ParkBool::lookupTable[] = {
    { "PFA", "ZENITH"},
    { "TRUE", "FALSE"},
    { 0, 0 } };


/* Using the value*/
ParkBool Position(DitsGetArgument(), "ZENITH",1,status);
if (Position)
...
```

GitEnum

This class implements an enumerated type which includes a member function to fetch values from an SDS structure using `GitArgGetS()`. This class provides a simpler interface to fetching enumerated values from strings in SDS structures. This is a virtual base class — the user must provide a class which inherits this class and provides implementations of the functions `SetValue()` and `Lookup()`.

The following code shows the definition of a class based on `GitEnum`, which interfaces to a enum with the values "Record}", "Dummy" and "Glance".

This particular example uses the constructor to do the get, hence it bans the default constructor by making it private. This is purely an issue for this particular example. You could provide an interface to the Get function and allow them. By making the `recEnum` enum definition public and providing an operator which returns the value, you could switch on this type instead of using the "Is" series of functions.

Example 17.9 Using GitEnum

```
class RecType : public GitEnum {
    private:
    /* enum possibilities */
        enum recEnum { Record=0, Dummy, Glance, Invalid };
```

```
      /* value contains the actual value */
         recEnum value;
      /* Lookup table returns the list of strings */
         static const char * const lookupTable[];
      /* Conversion operator, given enum, return an int */
         operator int() const { return ((int))value));

      /* return the lookup table address. Used by */
      /* GitEnum::Get */
         const char * const * Lookup() { return lookupTable;};

      /* Set value, used by GitEnum::Get */
         void SetValue(const unsigned int i) {
               if (i >= Invalid)
                     value = Invalid
               else
                     value = (recEnum)i;
         }
      /*
       * Since our constructor does a get, we prohibit the
       * default constructor and and assignment operators
       */
         RecType();
         RecType& operator=(const RecType &);
      public:
      /* The constructor - does a get   */
         RecType(
               const Sds& Id,
               const char * const Name,
               const int Position,
               StatusType * const status
               const char *Default = "RECORD",
               const int Flags=Git::Upper|Git::Abbrev) {

               Get(Id,Name,Position,status,Default,Flags);
         }
      /* Tests for enumerated values     */
         bool IsRecord const { return (value == Record);}
         bool IsDummy const { return (value == Dummy);}
         bool IsGlance const { return (value == Glance);}

};
/* Define static item declared above */
const char RecType::lookupTable[] = {
        "RECORD", "DUMMY", "GLANCE", 0 },

/* Using the class */
RecType recType(DitsGetArgument(), "RECORD_TYPE", 1 ,status);
if (recType.IsRecord())
...
```

GitInt and GitReal

These classes implement integer and floating point types respectively which include member functions to fetch values from an SDS structure using `GitArgGetI()`/`GitArgGetD()`. These classes provide simpler interfaces to

fetching integer and real values from Sds structures. (Note that at present, the implementation of these types is probably not complete, not all integer/real operations can be performed, although you can convert them to int/double as appropriate.

You can use these classes directly, in which case there are no limits to the range of the value read other then what can be represented in the relevant type. A more common usage is to sub-class this class in order to limit the range.

The following code shows the definition of a class based on `GitInt`, which interfaces to an integer range limited to between 1 and 100000. GitReal works in an almost identical way.

Example   17.10   Using   GitInt

```
/*
 * Repeat mode count integer
 */
class RepCount : public GitInt {
    private:
        virtual const long int * Range() { return range; }
        static const long int range[];
    public:
    /* Constructor with automatic get */
        RepCount(
                const SdsId& Id,
                const int Position,
                StatusType * const status,
                const int  Default = 1,
                const int Flags = Git::KeepErr) {
                GitInt::Get(Id,"REPEAT_COUNT",Position,
                            status,Default,Flags);
        }
    /* Simple constructor, defaulting to a value of 1 */
        GctRepCount(const long int def = 1) : GitInt(def){}
    /* Get function */
        void Get(
                const SdsId& Id,
                const int Position,
                StatusType * const status,
                const int Default = 1,
                const int Flags = Git::KeepErr) {
                    GitInt::Get(Id,"REPEAT_COUNT",Position,
                            status,Default,Flags);
        }
    /* Pre-decrement operator */
        GctRepCount operator--() {
                long int value = *this;
                *this = GctRepCount(value-1);
                return value;
        }
};
/*
 * Define static item defined above
 */
const long int GctRepCount::range[] = { 1,100000};
```

```
/* Using the Class */
GctRepCount RepeatCount(id,6,status);
```

DCPP

The DCPP (DITS C Plus Plus) set of classes implement a C++ interface to the DITS task communication functions. The scheme implemented allows you to send a message specifying callback functions to be invoked when responses are received. You can specify particular callback functions to handle particular responses.

We define a "Thread of control" as being the handling of response messages in relation to a message sent by our task. Assuming the thread of control is setup correctly, your callback functions will be invoked automatically. The callback routines return a code which indicate whether of not they are expecting further messages on the current thread of control.

The basic message sending call is `DcppTask`, which hides much of the work required to communicate with another DRAMA task. It can even hide an automatic load operation within a Get Path operation. An additional class — `DcppMonitor` — can be used to manage parameter monitors.

There are two types of control thread. First are normal DRAMA actions. In this case, you should install a variable of class `DcppHandler` to manage action rescheduling. This class will automatically invoked your callback handlers when messages arrive, only causing your action to complete when a callback handler indicates no more rescheduling is required.

The other type of control thread is `UFACE` context routines. Only people writing user interface code need be concerned with these threads. See `UFACE` Chapter XXXX and DITS_SPEC XXXX for more details on `UFACE` context. To use `DcppTask` based communications from `UFACE` threads, you should use the `DcppUfaceCtxEnable()` routine in place of the routine `DitsUfaceCtxEnable()`.

A task communication example

This example shows a very simple example of communication with a task using the `DcppTask` class. Below is an extract from the source code. All this example does is get the path to a task and send an obey to it.

Example 17.11 Using Dcpp

```
static DcppHandlerRet StartObey(                              ❶
                           DcppVoidPnt ClientData,
                           StatusType * const status);

static DcppTask ticker("TICKER",0,"DITS_LIB:ticker");          ❷
static DcppHandler Handler;                                     ❸
```

```
static void DpRun(StatusType * const status)                    ❹
{
    Handler.Install(status);                                    ❺
    if (ticker.GetPath(status,StartObey) == DcppReschedule)
    {
        DitsPutRequest(DITS_REQ_MESSAGE,status);
    }
}

static DcppHandlerRet StartObey(                                ❻
                            DcppVoidPnt ClientData,
                            StatusType * const status)
{
    return(ticker.Obey("TICK",status));
}
```

At ❶ is the prototype for the `StartObey()` function. This function is defined later and used as the "Success Handler" for the `GetPath()` operation ie., it is invoked when the `GetPath()` operation succeeds and only when it succeeds. At ❷ we see the definition of a `DcppTask` object named `ticker`. This is specified with a name of ``TICKER'' - the name the task is expected to have registered under if it is already running. A location of 0, (the null pointer) indicates we expect the task to be on the local machine and will load in on the local machine if it is to be loaded. The third argument specifies the file from which the task is loaded if it is not already running.

At ❸ we define a `DcppHandler` object. We don't specify any callback routines, which means that any errors on of completion subsidiary actions will cause the action to complete.

At ❹ is the definition of the `DpRun()` function. This function is an Obey Handler, which would be specified in a `DitsActionsType` variable passed to `DitsPutActions()`. An obey message will result in this function being invoked. At ❺ we install the handler, which will result in the `DcppHandler class` handling future reschedules of this action. We are a bit lazy in that we don't `DeInstall()` this object at any stage. This is OK as a `DeInstall()` is only required if we are to continue the action under our own control.

On the next line, we initiate a `GetPath()` operation on the `ticker` task object. We specify `StartObey()` as the success handler. By not specifying an "Error Handler", the action will complete on error. Now a `GetPath()` operation which returns with status ok will always result in a message transaction and it will therefore return the value `DcppReschedule`. So unless an error occurs, so we immediately request a reschedule. `GetPath()` automatically tries to load the task using the information available to it if it cannot find the task.

❻ is the definition of the `StartObey()` handler. This will be invoked when the `GetPath()` operation completes successfully. In it, we simply send the Obey message. Note that this routine is invoked in the context of the action by a routine in the `DcppHandler` class. `StartObey()` should return DcppReschedule if anything it

does requires the action to be rescheduled, which is also what `Obey()` will return if it works, so we just return the value of `Obey()`. No success, trigger or error handlers are specified which will result in the action completing when the Obey's completion message arrives. The previous `DcppHandler` object remains installed until the action exits.

By default, a `DcppHandler` object will keep track of one thread of messages. For example, the `GetPath()` operation followed by the `Obey()` message. If you wish to have multiple threads active at one time, for example, if you start a sequence of `GetPath()` operations, each followed by an `Obey()`, you should indicate to the `DcppHandler` object that there are multiple threads. For each thread after the first, you should invoke the `DcppHandler` increment operator (either pre or post increment). The `DcppHandler` object will then continue to reschedule the action until all threads have returned `DcppFinished`.

A simple example of parameter monitoring.

This example shows how to use the Monitoring class. It sets up a monitor of the parameter "PARAM1" in the task "TICKER'", defined in the previous example.

Example 17.12 Dcpp and Parameter Monitoring

```
static void DpMon(const char * const name,                    ❶
                  const SdsCodeType type,
                  const DcppVoidPnt value,
                  const DcppVoidPnt ClientData,
                  StatusType * const status)
{
    MsgOut(status,"Parameter %s changed",name);
}


DcppMonitor Monitor(&ticker);                                 ❷
DcppHandler Handler2;                                         ❸

static void DpTest (StatusType *status)                       ❹
{
    Handler2.Install(status);

    Monitor.Monitor(DpMon,0,0, false,1,status,"PARAM1");      ❺
    DitsPutRequest(DITS_REQ_MESSAGE,status);
}


static void DpTestKick(StatusType *status)                    ❻
{
    Monitor.Cancel(status,DcppTask::DiscardResponse);
}
```

At ❶ we have the definition of the routine that will be invoked automatically when the parameter value has changed. In this example, it only outputs a simple message. At ❷ we have the definition of the monitor object. As its argument we have specified the address of a `DcppTask` object named `ticker` that would have been declared previously. At ❸ we define a `DcppHandler` object.

❹ is the definition of the action handler routine that will be invoked to do the work. After installing the handler, we start the monitor at ❺ specifying the routine `DpMon()` to be invoked when the parameter changes and `PARAM1` as the name of the parameter to monitor. We assume at this point that some where a
`GetPath()` operation has already been performed on the `DcppTask` object named `ticker`. That is all there is to it. The rest of the work is done within the `DcppMonitor` and `DcppHandler` classes.

The one thing left to do is to cancel monitoring if necessary. In this example, a kick handler is provided at ❻. All we need to is send a cancel to the monitor. By specifying the `DiscardResponse()` routine as the success handler, we are saying we want to ignore any response. The default kick routine rescheduling (no effect on the Obey context) is sufficent.

A more complex example.

A more complex example and complete documentation for Dcpp and the other C++ DRAMA classes are found in document 12 - XXXX.

Efficiency Considerations.

It is reasonable to compare the efficiency of the C and C++ interfaces to DRAMA. A this stage, this is largely done using internal knowledge of the code concerned.

The `SdsId` class adds one item (byte or word size, depending on the compiler and target) to each SDS id is maintains. This is used to maintain the flags which indicate if the item should be delete, freed-ed etc. An additional word is required for the virtual function lookup pointer. Constructors must add a small amount of code to set these items up, but otherwise, there is generally no overhead added to the run time efficiency of the underlying Sds calls.

The `Arg` class adds no overhead over the SdsId overhead required for each SDS id.

The `Sdp` class adds no overhead over the C interfaces, since all `Sdp` class calls are inlined to the equivalent C call.

The `GitBool`, `GitEnum`, `GitInt` and `GitReal` classes add at least one pointer to each data item — used to find the virtual function lookup table. Other then this, they are probably not any more inefficient then using the equivalent `GitArg*()` functions directly. In the case of `GitEnum`, most of the work is in the declarations, not in the resulting code. I believe the example code given `RecType` would use no more runtime code then a direct call to the `GitArgGetS()` function and the associated checks. You do through get much more compile time checking.

For `Dcpp` and assoaicated classes, a well-defined amount of overhead is added to the initiation of each message being the allocation and filling in of a structure to maintain details associated with a transaction. This structure is dynamically allocated. If this

proves a problem, then the default allocator could be replaced with a more appropriate algorithm.

The overhead added to handle rescheduling is probably not much higher then that normally required to sort through the possibilities in C code.

18.Signals, Interrupts, Alternative Input Sources

Overview

In this chapter we examine the facilities provided by DRAMA which allow communication between DRAMA and other sources of input a program may have. The possible other sources of input are dependent on the operating system involved, but there are normally two classes. The first is asynchronous interrupts, generated in either software or hardware. Here we include Unix style "signals". The second class is notification of data availability from the file system.

But before we look at these facilities, we shall first examing the default handling of Signals and associated operations.

Signals Etc. default handling

.Most operating systems have some way of asynchronously shutting down a program either in response to some error or in response to a demand from the user, say a Ctrl-C under Unix. In order to ensure that resources are released and other DRAMA programs are told about a DRAMA task disappearing, DRAMA must be told about such an event before the program disappears. The technqiues involved are system specific and can have affects on other parts of a complex system, so I will describe them.

VxWorks task deletion.

Under VxWorks, the key to DRAMA's handling of this problem is the "task deletion hook". DRAMA installs such a hook the first time a DRAMA program starts up. And then every DRAMA task which starts up registers with the hook. Each task removes itself from the register when it is shutdown.

When a DRAMA task is forceably deleted, by the VxWorks function `taskDelete()` or the shell command "`td`", the hook is invoked *in the context of the task which triggered the deletion*, not the context of task being deleted. The deletion hook "makes itself" into the task being deleted and does the DRAMA cleanup for that task.

Fortunately, there is no way to kill a task under VxWorks without deletion hooks being invoked.

VMS Task Deletion

The VMS operating system provides good support for handling of a task dying. It does this in two ways. First, almost all resources are correctly cleaned up when a task dies. Secondly, an Exit handler can be supplied which will be executed in most cases. Only if a task is expliclty "stopped" using the `DCL` "`STOP`" command or the XXX system service is this exit handler not run. Even here, you could first it to be run by "Installing" the program and even if you don't the automatic cleaning up resources is still done. See the `SYS$DCLEXH()` VMS system service for details.

WIN32 Task Deletion

Under the WIN32 (Windows NT and Windows 95/98) based operating systems, the core of DRAMA resides in a `DLL` (Dynamic Link Library or Shared Library). A `DLL` can be notified each time a program start's up and shutdown. DRAMA makes use of this notification to clean itself up. This approach will fail if the XXX WIN32 routine is used to destroy a task since in that case the DLL is not notified. Since this routine is the normal way to delete another program, it presents a problem. DRAMA own task deletion routine — `DitsDeleteTask()` allows you to avoid this problem. When called with the "FORCE" flag set false, `DitsDeleteTask()` will insert a new thread into the target task. This new thread immediately deletes the target task using XXX,WIN32 routine, which ensures the `DLL` is notified.

Unix Signals.

Unix presents real problems. Although you can specify an exit handler routine, using `atexit()` or XXX, these are implemented within the C run time library only, not by the system. As a result, they rely on the program calling `exit()` or returning from main. If the program dies as a result of anything else, DRAMA would not be notified. Unix signals are the normal way that a program is shutdown without calling `exit()`. Signals can be generated internally to the task, say due to an address or divide by zero error. The can also be generated externally using the `kill()` system service.

DRAMA attempts to handle all Unix signals which may cause a program to exit. When `DitsAppInit()` is invoked, DRAMA uses system calls to intercept each of these signals, saving any existing signal handler for each signal. When a signal is received, the DRAMA signal handler does two things. First, it tries to shutdown DRAMA. Secondly, it tries to do whatever the original signal handler would have done, core dump for example. Since one process can't do both of these, it most fork in the signal handler with the original process shutting down DRAMA.

Note that if you use "`kill  -9`", to kill a task, then the DRAMA exit handler is never invoked. Always try a plain "kill" first. Due to problems resulting from "kill -9" and sometimes from program crashes, a utility is needed to tidy up after DRAMA programs in these cases. This is the "`cleanup`" command. This program has several options, but in it's default mode of operation, will kill any running DRAMA program and tidy up resources allocated by those programs and any which had previously died without cleaning up after themselves.

After calling `DitsAppInit()` you can override this for any particular signal using the normal Unix signal features.

Turning off the default DRAMA handling.

As you have probably noticied, the above techniques are complex. You may find or may suspect that they are interacting with similar techniques used by your program or by libraries required by your program. If you wish, you may tell DRAMA not to install its exit handlers. This is done by specifying the `DITS_M_NOEXHAND` flag to

`DitsAppInit()`. Having done this, you should try to ensure that `DitsStop()` is invoked before the program shuts down.

Interactions with the Master Task.

If a task is loaded using the DRAMA task loading facilities, such as the `DitsLoad()` routine, and the Master Task is used to do the load (which is the normal approach), then if a task crashes without cleaning up after itself, the Master Task will tidy up on its behalf. This provides one extra layer of protection against resources being left around after a program has crashes. This occurs regardless of the setting of the `DITS_M_NOEXHAND` flag, but requires that a program has been loaded using the DRAMA facitilites.

DRAMA and Asynchronous Events.

Most complex real time systems will require interactions between the main message loop of a program and asynchronous events. The type of asynchronous depends on the operating systems, but includes Unix Signals, VMS ASTs, WIN 32 Threads and asynchronous I/O and VxWorks Interrupt handlers. The way these style of interactions is handled under DRAMA is that the author provides the routine which responds to the event at a low level, such as an Interrupt ISR. From within this routine, you can send a message to the DRAMA event loop, causing an action to be rescheduled.

For the purpose of this discussion, we assume the worst case system, a VxWorks Interrupt Service Routine (ISR). This is the worst case because of the common address space used by the different tasks in a VxWorks system and because it is a real ISR, which may crash the system. Having said this, the DRAMA code applies just as well to the other operating systems supported.

The DRAMA Interrupt handling stuff assumes

1. A DRAMA action is used to trigger some physical event which later causes an interrupt. The DRAMA action goes to sleep and is to be rescheduled when the interrupt occurs.

2. The DRIVER which actually handles the interrupt can be told the address of a routine to be invoked as an application specific ISR.

   There must also be some way of getting some "client data" item passed to the ISR. This can just be via a global memory item but we normally use features in our drivers which allow an argument to passed to the ISR.

3. The ISR routine handles time critical aspects and then "signals" DRAMA that the interrupt has occurred indicating which DRAMA action is to be rescheduled.

4. The DRAMA action is then rescheduled and handles the less time critical apects.

5. In this scheme, the VxWorks driver for the hardware concerned knows nothing about DRAMA. All it need do is provide the features mentioned in 2. We use normal VxWorks I/O calls to do the actual I/O.

Lets consider a dummy example, for which we write some pseudo code indicating the DRAMA calls required. Lets assume the example involves moving a filter wheel and there is an action to move the filter wheel named "MOVE_FILTER".

We assume low level VxWorks driver has been provided which provides the following operations via a low level library.

```
void FilterWheelCommenceMoveTo(int position);
void FilterWheelInstallISR(void *routine,
                           void *clientData);
void FilterWheelClearInterrupt();
int FilterWheelGetCurrentPos()
```

Remember here, DRAMA does NOT provide the above routines. The example requires that the action MOVE_FILTER starts the filter wheel moving and then wait for the interrupt which indicates the wheel is in position. The action then tidies up after the interrupt and completes.

Example 18.1   ISR Routines

```
/*
 * This routine implements the first stage of the action
 *   MOVE_FILTER.
 */
void MoveFilterAction(StatusType *status)                         ❶
{
    int Where;
    if (*status != STATUS__OK) return;
/*
 * Get and validate the action argument which indicates the
 * required wheel position  (routine provided somewhere else in
 * application)
 */
    RequiredPosition(&Where,status);

    if (*status == STATUS__OK)
    {
/*
 *      Clear any outstanding interrupts and Install ISR
 *
 *      Note the use of DitsGetTaskID() as the client data
 *      item. This  is needed by the ISR.
 */
        FilterWheelClearInterrupt();                             ❷
        FilterWheelInstallISR(MyISRRoutine, DitsGetTaskID());
/*
 *      Start the move operation.
```

```
 */
        FilterWheelCommenceMoveTo(Where);
    }
/*
 * Wait for the filter wheel move to complete.  Function
 * "FilterMoved" will then be invoked as the next stage
 * of the action.  Note, you can also set up a timeout here, say
 * by calling DitsPutDelete() or GitPutDelay().
 */
    DitsPutObeyHandler(FilterMoved,status);                        ❸
    DitsPutRequest(DITS_REQ_SLEEP,status);
}


/*
 * ISR routine installed by the above action routine.  Invoked
 * in interrupt context by the DEVICE DRIVER.
 */
void MyISRRoutine(void *ClientData)                               ❹
{
    DitsTaskIdType SavedID;
    StatusType status = STATUS__OK;

/*
 * Do any work required by the driver.
 */
    FilterWheelClearInterrupt()                                   ❺

/*
 * Enable DRAMA task context.  This is required before
 * calling DitsSignalByName().
 */
    DitsEnableTask((DitsTaskIdType)ClientData, &SavedID);         ❻

/*
 * Call DitsSignalByName().  This actually sends a message
 * to the task causing the action to be rescheduled.
 *
 * The first argument is the name of the action to be signalled.
 *
 * The second is an SDS id, which can be fetched in the action
 * routine by DitsGetArgument(), but in this case, we CHEAT
 * by passing the integer value returned by the function.  This
 * is ok as long as the  action code invokes DitsArgNoDel().
 * See FilterMoved() implementation.
 *
 * Note that you cannot create or modify the structure of SDS
 *  items in an ISR, although you can use SdsPut/SdsGet.
 */
    DitsSignalByName("MOVE_FILTER",                               ❼
                  (SdsIdType)FilterWheelGetCurrentPos(),
                  &status);

/*
 * DitsSignalByName will fail if the action is not active
 * (waiting to be rescheduled) returning the status code
 * DITS__ACTNOTACT.
 *
 * It will also fail if the action name is invalid, returning
```

```
 * DITS__UNKNACT.
 *
 * IMP level failures are also possible but less likely.
 *
 * About the only place you can report such failures is to the
 * VxWorks console, using the VxWorks routine logMsg.
 */
    if (status != STATUS__OK)
    {
        logMsg(
            "MyISRRoutine:DitsSignalByName failed,status= %x\n",
                status, 0, 0, 0, 0, 0);
    }
/*
 * Restore the task context.
 */
    DitsRestoreTask(SavedID);                                   ❽
}


/*
 * Second stage of action MOVE_FILTER.  Invoked after interrupt
 * occurs.
 */
void FilterMoved(StatusType *status);                          ❾
{
    int where;
    if (*status != STATUS__OK) return;

    if (DitsGetEntReason() == DITS_REA_ASTINT)
    {
/*
 *      The ISR passed the new position to us as an integer
 *      argumentto DitsSignalByName().  We can fetch this
 *      using DitsGetArgument().
 *
 *      DITS expects arguments to DitsSignalByName() to be
 *      SDS items and will delete these SDS items when this
 *      action entry returns.  Since in this example it is not
 *      an SDS item, we must tell DITS not to try to delete or
 *      free it by calling DitsArgNoDel().
 */
        where = (int) DitsGetArgument();                      ❿
        DitsArgNoDel(status);


        ...
    }
    else
    {
/*
 *      Should not happen in this example
 *      But if we have enabled a timeout  for example
 */
        ;
    }
}
```

At ❶ we have the action handler routine for the MOVE_FILTER action. In this example, it first grabs and validates the desired filter wheel position in the routine RequiredPosition(). This is assumed to be implemented somewhere else in the application, probably by calling GIT library routines.

At ❷, we use the driver provided routines to clear the interrupt, (which many real world examples require) and to install our ISR routine. The ISR routine is MyISRRoutine(). The second argument to FilterWhellInstallISR() is a "ClientData" item to be passed as an argument to the ISR routine. In this example, we need to pass the value of DitsGetTaskID(). DitsGetTaskID() returns a value which is needed in the ISR routine to enable the use of DRAMA calls.

At ❸, we reschedule the action to await the interrupt.

At ❹, we have the definition of the actual ISR routine. Note the "ClientData" argument which is used to pass the value of DitsGetTaskID(). When writing such a routine you are heavily restricted in what you can do. Under VxWorks, the task context may be of any task in the system. As a result, you can't just use DRAMA routines since they rely on the task context to access global data items.

In many ISRs, there will be some work which must be done to clear the interrupt. In this example, we call FilterWheelClearInterrupt() at ❺.

After this, we need to send a message to our DRAMA task. Before we can do that, we need to set up the task context such that DRAMA routines can be used. This is done at ❻ using DitsEnableTask(), which takes as its first argument the value returned by DitsGetTaskID() found in "ClientData". You must supply the address of a variable, in this case SavedID, where the current task context is saved.

Now you can send the message to your DRAMA task. This is done at ❼ by calling DitsSignalByName(). You specify the name of the action to signal and an argument for the action. You can supply an SDS id which becomes the values returned by DitsGetArgument() when the action is scheduled. In this case, we cheat by use specifing an integer value here. This is safe as long as the action uses DitsArgNoDel() when it is reschedule. If you do specify an SDS id, you must not delete the SDS item or free the id as this is normally done by the action reschedule code.

After sending the message, we must restore the task context to what it was on entry to the ISR. This is done at ❽ by DitsRestoreTask().

Finally, we have the second stage of the MOVE_FILTER action at ❾. When an action reschedule is triggered by DitsSignalByName(), the reason will be DITS_REA_ASTINT.

At ❿, the argument supplied to DitsSignalByName() is accessed. In this, example, we were just passing an integer value, rather then an SDS id, so we call

`DitsArgNoDel()` to ensure DRAMA does not try to delete an SDS item with the same integer ID value as the item we are passing.

That's it. One note about the the DITS signalling routines. There
are two real routines `DitsSignalByName()` and `DitsSignalByIndex()`. The former takes an action name and the later an action index. The index of an action can be fetched using `DitsGetActIndex()`. Using the index is faster but means there is another item you have to pass to the ISR. In addition, using the index allows spawnable actions to be signalled.

In addition, older code uses `DitsSignal()`. This is just a different name for `DitsSignalByName()`.

As mentioned above, the example is a VxWorks based example,the most restrictive case. In operating systems where the asynchronous events occur in the same memory context as the main line code, the use of `DitsEnableTask()` and `DitsRestoreTask()` are not necessary, although they do no harm.

Alternative   Input   sources.

INCOMPLTE

Example   18.1   Alternative   InputSources

```
/* List the item "fieldData" within the specified SDS file */
}
```

19 Determining DRAMA Buffer Sizes

Overview

This chapter attempts to answer one of the most complicated questions in DRAMA, how to set DRAMA message buffer sizes.   First, we will have a look why this scheme is used. We will then look at the scheme itself and then how to select your buffer sizes. We also look at the error messages and other details related to buffer sizes. This chapter is surprisingly large. This is because it attempts to give all the gory details, and is complete with diagrams and examples. Hopefully, your need only familiarise yourself with what is here and only need read the full details if you hit problems.

Please note that the shaded parts of this chapter represent things which I intend to change by changing DRAMA.

Message   Sending   Techniques

There are many different techniques you might use to send a message from one task to another. Factors involved in the selection of the technique to be used include

- The facilities provided by the computer OS.
- The locations of the tasks involved, i.e. on the same or different computers.
- The speed required.
- The sizes of the messages to be sent.
- The number of messages to be sent.
- The expected reliability of the underlying message transport mechanism.

Having said this, all the possibilities divide into two broad classes, these are "Rendezvous" based techniques and all the rest. Rendezvous based techniques rely on both tasks involved meeting at a pre-defined point. Only when both tasks are at the right point can message passing occur. The major benefits of this technique are reliability and cheap memory management.

For example, consider two tasks A and B. If task A wants to send a message to task B, task A must call a message sending routine. For task B to receive the message, it must call a message receiving routine. The particular protocol determines what happens if say A tries to send when B is not waiting for the message, but in many cases, A would just block until B is ready to receive the message.   If something goes wrong in the message transfer, both tasks can immediately and reliably receive an exception. Memory management is normally easy, the receiver knows how much data it is to receive and can allocate memory for it as part of accepting the message.

Rendezvous based techniques could well be the only techniques you would consider in situations where reliability is ultra important, such as where a failure would cause loss of life. It is probably for this reason it is the only technique available in ADA's tasking system.

But there is more to computing then nuclear power stations and fly by wire planes. The major problem with the rendezvous techniques is that the sending task has to wait for the receiving task to accept the message. If something goes wrong, or the receiving task is just busy, the sending task blocks. This often creates a poorly responding system which annoys the user.   So non-rendezvous techniques allow the sender to send the message then continue working. The receiver reads the message when it ready.

What do we lose with this approach - cheap and highly reliable error handling. If the receiver cannot process the message, another message needs to be sent to the sender to tell it of this.   But for systems which must be responsive to multiple external events, this is by far the better approach.   Note that rendezvous based systems can use threads or similar techniques to get around the problem, but they then have trouble with access to variables which must be used by both threads.

So DRAMA chose the non-rendezvous approach. In most such schemes, the sender writes the message to a buffer and then notifies the receiver that there is a message waiting.   The sender then continues with its job, possibly involving sending more messages to the same task. The receiver just reads the messages from the buffer and then marks the buffer as empty, allowing the sender to reuse the buffer.

DRAMA's Message   Sending   Technique

The particular technique used by DRAMA is the use of shared memory message buffers combined with a notification technique. Using shared memory allows a task to write a message directly to the receiving task's message buffers. In addition, DRAMA provides networking tasks which emulate the base facility when the tasks are on different machines.

The notification technique used is system and usage dependent, allowing DRAMA to choose the fastest technique which will do that job and to remain compatible with other systems with which DRAMA must work.   For example, on Unix systems, we use Semaphores or Message Queues or FIFO's. FIFO's are used on a task specific basis if compatibility with X-Windows is required (i.e. the notification technique must work with the `select()` system call). Otherwise, the fastest technique of either Semaphores or Message Queues is used.

The actual implementation of shared memory also depends on the operating system. Under Unix, mapped files are the default technique with the files being located in the directory indicated the environment variable `IMP_SCRATCH` (defaulting to the user's home directory. It is good idea if this points to a disk on a local machine, not an NFS disk). Under VMS, VMS global sections are used. Under VxWorks, it is simple

`malloc()` allocated memory (as all memory is global under VxWorks). Under WIN32, it is Mapped files located in the system virtual memory file directory.

The primary aim of this chapter is to examine the message buffer usage.

The Global Buffer Space

In the current version of DRAMA, all message buffers allocated within a task are allocated from the "Global Buffer Space". This is an area of shared memory created when the program calls `DitsAppInit()/DitsInit()`. This means you must have some prior knowledge about which programs will be communicating with your tasks and how much buffer space they will try to allocate. This is considered a flaw in the DRAMA implementation. A future version may remove the "Global Buffer Space" and allocate shared memory dynamically each time a path is allocated.

The Message Buffers

Figure 19.1 shows two tasks, A and B and the shared memory they have pre-allocated for use as message buffers. The amount of this pre-allocated memory is determined by the "Global Buffer Space" space argument to DitsInit() or DitsAppInit().
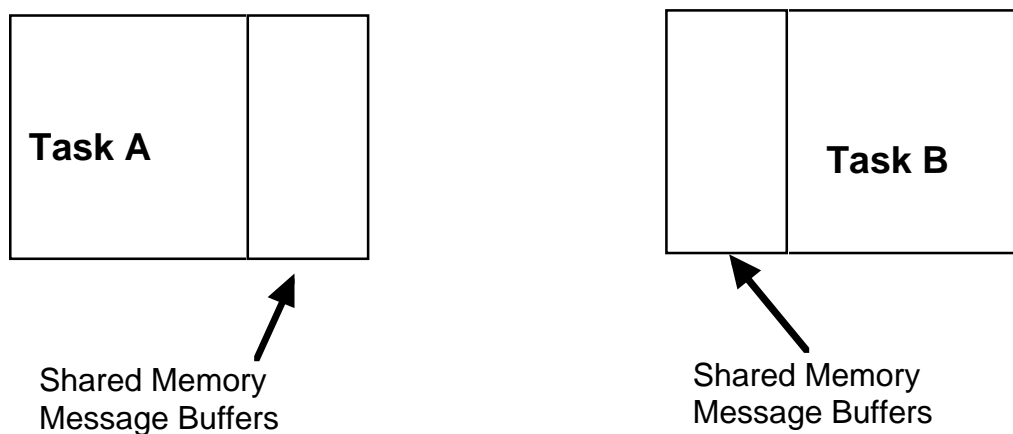
Figure 19.1 The Shared Memory Buffers

Now in order for task A to send a message to task B, task A must allocate some space in task B' shared memory space for those messages. Now task A only allocates what it needs, allowing other tasks to also talk to task B. Figure 19.2 shows this.

Space Allocated By
Task A

Task A

Task B

Shared Memory
Message Buffers

Shared Memory
Message Buffers
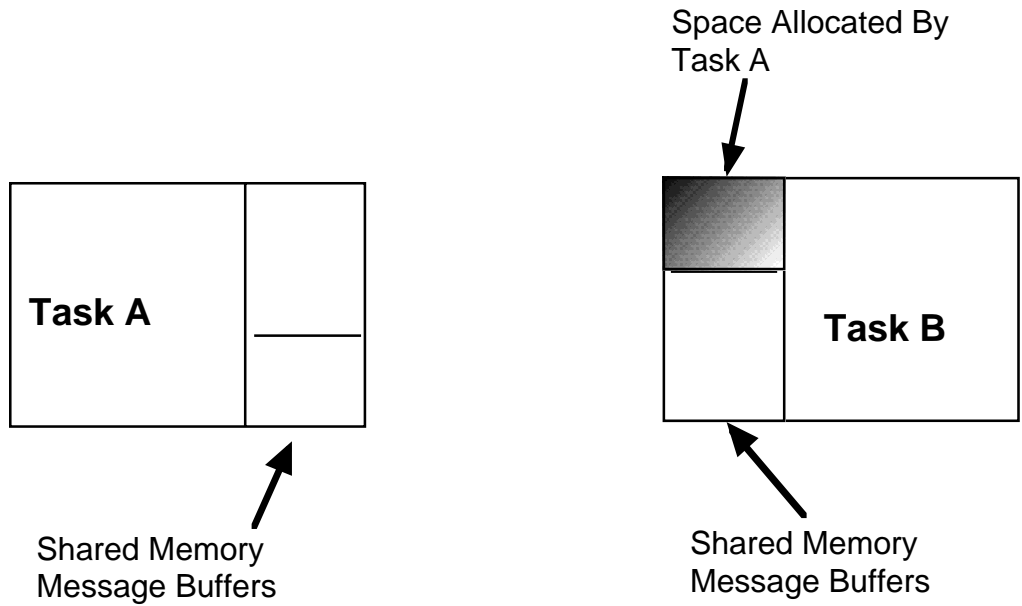
Figure 19.2 Allocation of space in task B by task A

Now whilst this allows task A to send messages to task B, it does not allow task B to respond. For task B to respond, it must have some space to write messages to task A. Figure 19.3 shows this.

Space Allocated By
Task B

Space Allocated By
Task A

Task A

Task B

Shared Memory
Message Buffers

Shared Memory
Message Buffers
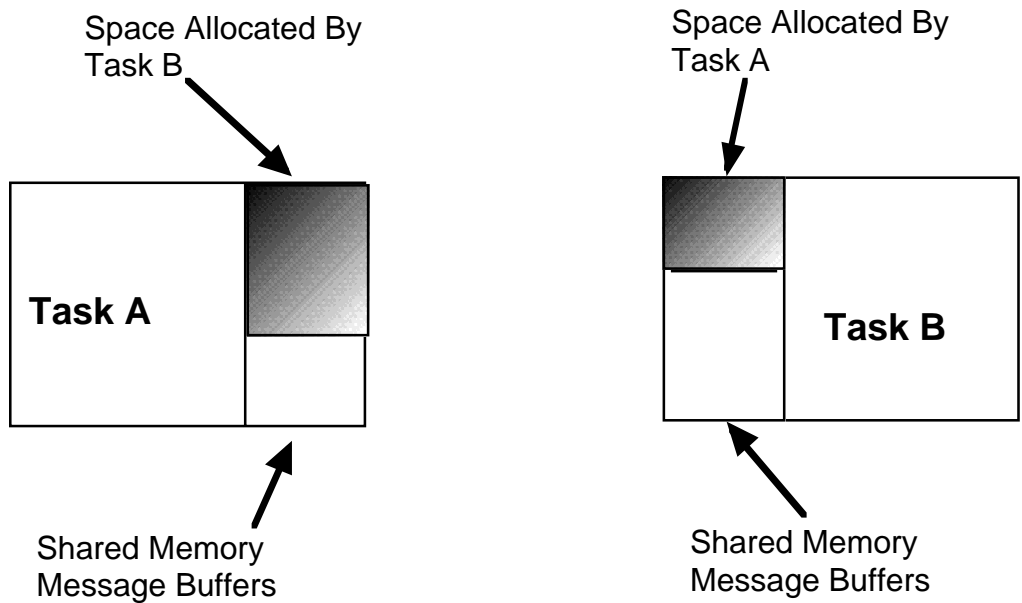
Figure 19.3 Allocation of space in task A by task B.

In this example, task B has allocated more space in A then task A has allocated in B. This buffer allocation is done by `DitsPathGet()`, which hides that details.   We will be examining the buffer sizing later.

Sending   Messages.

Having set up buffers for communication between two tasks using `DitsGetPath()`, we can send messages along the path. Most DRAMA messages involve a two-way flow of information. For example, in the sending of an Obey message, task A sends a message to task B, which will at some later time, send one or more relies. From the point of view of the message buffers, what happens when task A calls `DitsObey()` is that the message is written into the buffers in task B. This is shown by Figure 19.6.
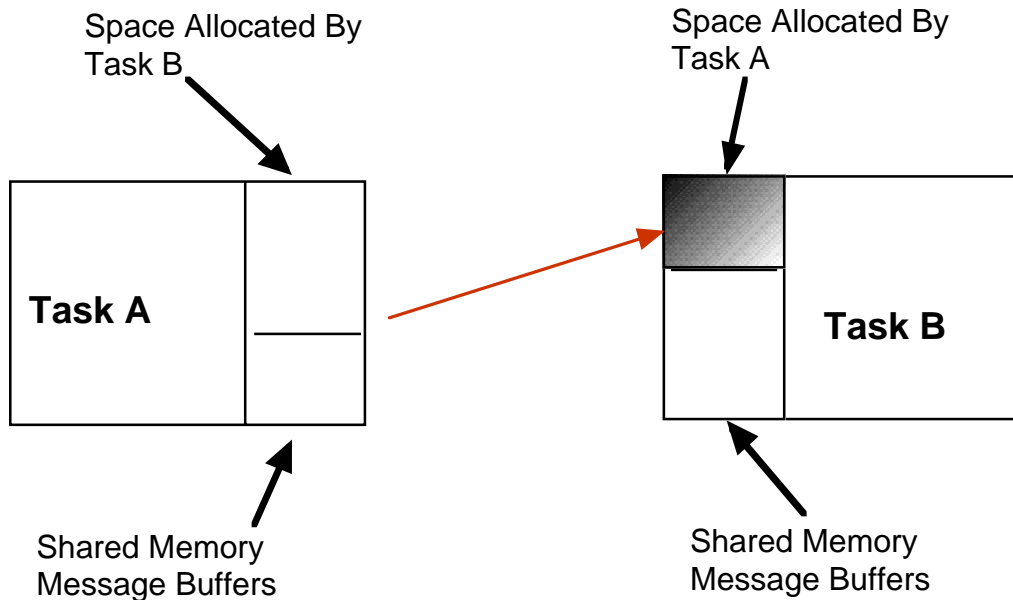
Figure 19.4 Sending a message from task A to task B.

If the message buffer is initially empty, the message sent can be any size up to the full size of the message buffer. Any space remaining in the buffer can be used to send additional messages whilst waiting for task B to process the first message. When task B reads the message, it does the initial processing of the message (the first stage of the action) and then releases the buffer space allowing it to be used again. Note that task B reads directly from the message buffer and any SDS argument to the Obey is accessed directly from the message buffer. This is the reason you must copy SDS argument structures if you want to keep them about after the action stage completes.

Task B will normally want to send one or more response messages back to task B. These responses may be `MsgOut()`, `ERS` or `DitsTrigger()` messages, as well as action completion responses. This is shown by figure 19.7.

Space Allocated By
Task B

Space Allocated By
Task A

Task A

Task B

Shared Memory
Message Buffers

Shared Memory
Message Buffers
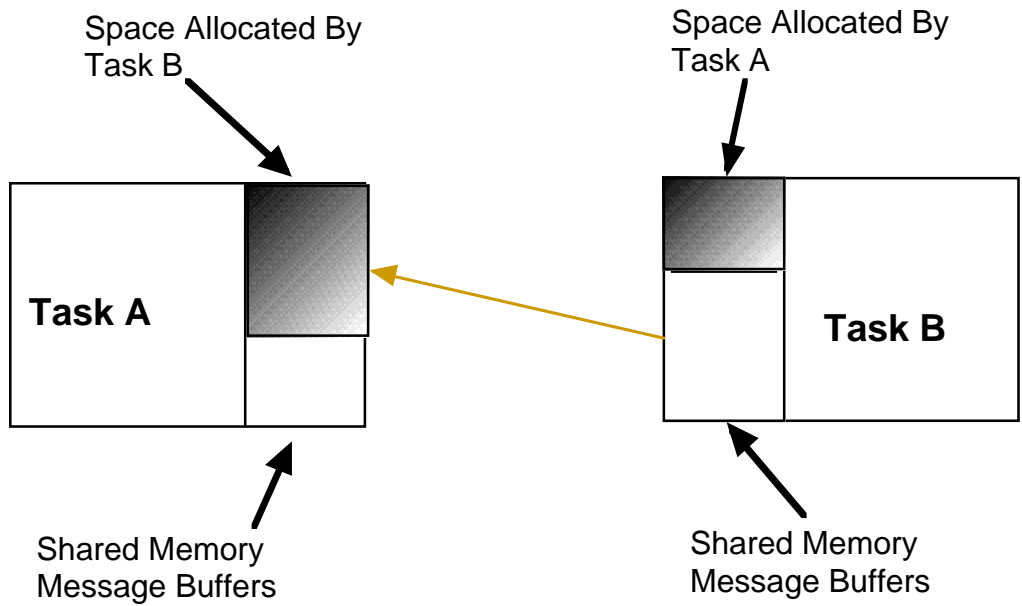
Figure 19.5 Sending replies back to task B.

Once again, only the required space is used and the remaining space can be used to send other messages. An important thing to note is that communication between task A and B can only be done through the pre-allocated buffers set up between those tasks. If the buffer in question is full, they cannot use the remaining space in the shared memory. That is reserved for other tasks to allocate.

Figure 19.8 shows the allocation of buffers when a third task, task C, is also trying to communicate with task B. The scheme works exactly the same, but task C can only allocate from the space remaining in task B's global buffer space.
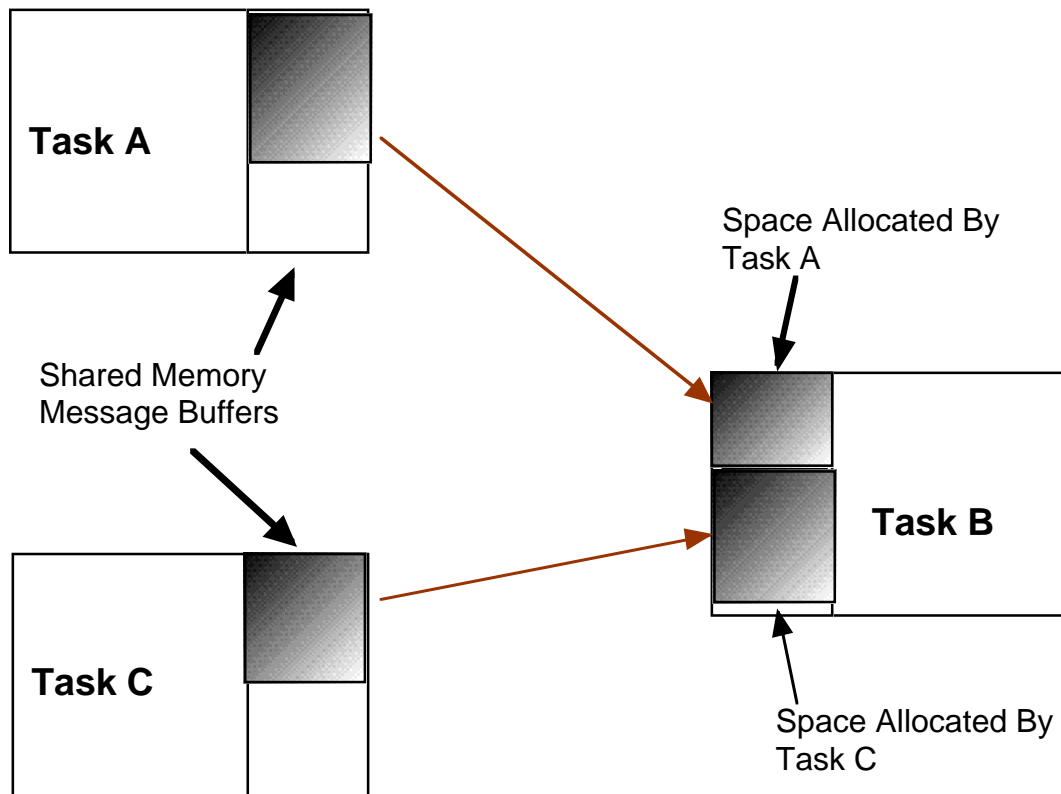
Figure 19.6 Allocation by task C in task B.

You can see that in this example, very little space is available in task B for a third task to commence communications with it.

Networking

When networking is involved, the protocol becomes more complicated, although this is largely hidden from the DRAMA Application programmer. In this case, the DRAMA networking tasks "transmitter" and "receiver" are involved. If task A and B are on different machines, then task A will actually allocate buffer space in the "transmitter" task on its machine and it is the "receiver" task on task B's machine which allocates the space in task B.

Figure 19.9 shows what happens when task A and B are on different machines. You will note that the two transmitter tasks have their own global buffer space which allocated by the other tasks. When task A calls `DitsObey()` to send a message to task B, it actually sends the message to the local "transmitter" task. Transmitter transparently forwards the message to the remote "receiver" task that in turn is responsible for passing it to task "B".

This scheme was chosen to avoid every DRAMA task having to have the overheads involved in networking communication. It also means that a send to a local task or a remote task appear identical to the sending and receiving tasks, other then the network delays involved.
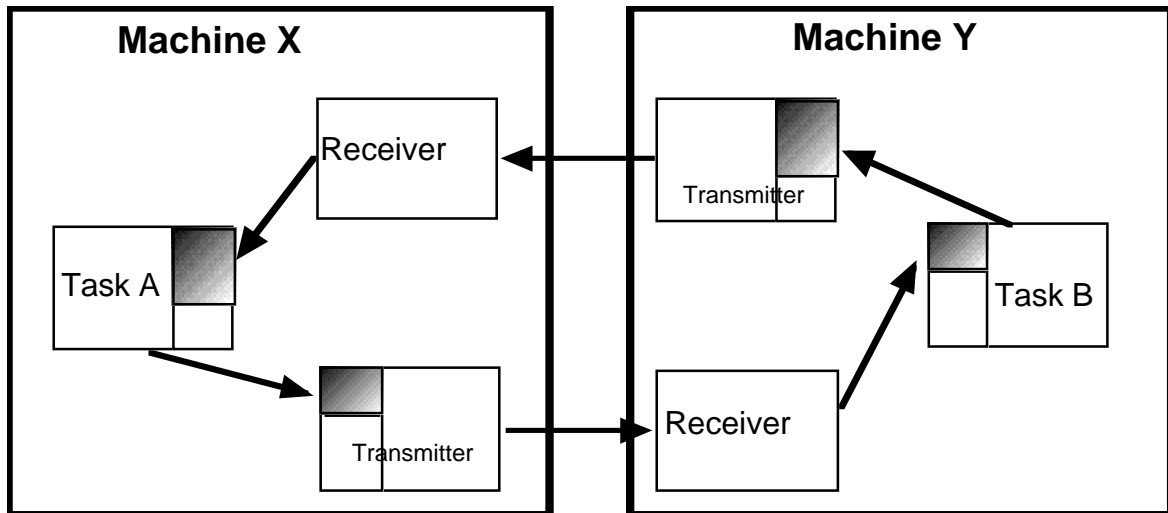
Figure 19.7 Tasks A and B on different machines

Buffer Sizes.

One of the hardest jobs in DRAMA is selecting the correct buffer sizes. Part of the problem is determining what values are to be set.

First there is the global buffer space in each task, this is the amount of shared memory allocated from the system, and from which the message buffers for a particular task will be allocated. For user tasks, this value is set by the "bytes" argument to `DitsAppInit()/DitsInit()`. A good initial value for this is item 20000. If your task is very busy, has multiple paths to other tasks or receives large messages, such as images, it may need to be much higher. Note that not all of this space is available for your message buffers. You should assume about 4000 Bytes is used by the system.

If you are sending large amounts of data across the network, you may need to increase the global buffer space allocated by the "transmitter" task involved. By default, each transmitter task allocates 1Mb of global buffer space. You can change this value by setting the system symbol (Environment Variable/Logical Name, depending on the OS) named `IMP_NET_KBYTES` to the number of Kilo-Bytes you wish transmitter to allocate. This should be done before you start the DRAMA networking on the machine where you want to change the allocation. For example, to up the value to 2Mb -

```
setenv IMP_NET_KBYTES 2048
dits_netstart
```

For each path between two tasks, the amount of space allocated for that path is determined by the `info` structure argument to DitsPathGet(). This structure has four elements of interest, `MessageBytes, MaxMessages, ReplyBytes` and

`MaxReplies`. Consider the example in figure 19.3, which was set up by task A calling DitsPathGet(). The space allocated by task A in task B, in bytes, is given by

```
(info.MessageBytes + IMP_MSG_HEADER_SIZE) * info.MaxMessages
```

The value of `IMP_MSG_HEADER_SIZE` depends on the architecture and the revision of IMP. It provides sufficient space for the message header added by the underlying IMP message system. The variable nature of this item is why you need two numbers to define the buffer size, rather then just the total space.   In the current version of IMP, `IMP_MSG_HEADER_SIZE` is roughly 32 Bytes. The idea is that you determine the size of the largest message you may want to send as well as the maximum number of them which may be waiting to be processed at any time. Having done that, the IMP layer can account for the size of it's own header in each message.

An important point here is these variables (`MessageBytes` and `MaxMessages`) are only used for this calculation.   Once the total space to be allocated is determined, only that total is used anywhere in the system. So, you can send messages of any size up to the total and you can have a larger number of smaller messages waiting for processing.

Considering figure 19.3 again, the space allocated by task B, in task A buffers, for reply messages is given by

```
(info.ReplyBytes + IMP_MSG_HEADER_SIZE) * info.MaxReplies
```

and all the other considerations mentioned above apply, except that there is a minimum value and some overhead which are applied automatically by the IMP layer. The minimum value is about 350 bytes, and the overhead is up to that value, depending a bit on the actual size.

There are two things about `DitsPathGet()` you should be aware of at this stage. First, only the first `DitsPathGet()` call for a path between two tasks is significant. It is at that stage that the buffers are allocated. As a result, if you make a later call to `DitsPathGet()` with different buffer sizes, they will be ignored. The only way to change the buffer sizes dynamically is to close the path using `DitsLosePath()` and then open it again. But beware of timing problems (losing a path whilst waiting for a reply on that path).   The timing problems are why this facility is not provided automatically by `DitsPathGet()`.

The second thing to beware of is that paths are inherently two-way. If after task A has got a path to task B, task B attempts to get a path to task A, that path will already exist and the buffer sizes have already been allocated.

If a task wishes to send messages to itself, the buffer sizes are set by the call to `DitsAppInit()`. See the `man/html` page for details. This is because DRAMA uses such a path itself and allocates it at that time. There is a default size of 2000 bytes for the self-path.

Sizes of Messages

So clearly one of the things you need to determine are the sizes of the messages your task will send. DRAMA has two classes of messages, GSOK messages and TAP messages. GSOK is an abbreviation for Get/Set/Obey/Kick, representing the four original named message types the user could send. Three other types have been added over time, the Monitor, Control and MGet (Multiple Parameter Get) messages. These messages are all sent using the `DitsInitiateMessage()` routine, although several have higher level wrap arounds, such as `DitsObey()`.

TAP messages are used for the replies to GSOK messages. They are sent by the `MsgOut()` and `DitsTrigger()` routines, the ERS package and as the action completion message. You never send one of these messages directly, but instead via these facilities.

Normally a message of either of these two classes consists of a header and a user supplied argument. The user-supplied argument is an SDS structure. So to determine the size of a message, you need to know the size of the header and the size of the argument structure. To help, DRAMA provides the `DitsGetMsgLength()` routine. This routine takes, as its first argument, a flag indicating which class of message you want the size of. The flag is set false for GSOK messages and true for TAP (reply) messages. Since the header of each class of message has the same size in all messages of that class, you don't need to indicate the type of message within each class.

The second argument to `DitsGetMsgLength()` is the SDS id of an argument to be sent with the message. Note that all the data elements in this structure should be defined, if they are to be defined when the message is sent. (See the SDS manual for details about "defining" data items, but basically this means you should have put data into the structure, not just have created it).

The third argument to this routine is the address of a variable to hold the number of bytes that will be required to send the message. The fourth argument is the normal modified status argument.

How do you use this routine? One approach is, when building an application, to add a temporary call to this routine with the largest structure that you intend to send. Then print out somewhere the result. This allows you to size the message buffers correctly. If you only have to consider messages without SDS argument structures, then the sizes are always the same for each class. For GSOK messages in the current version of DRAMA (1.2), this value is 304 bytes. For TAP messages, this is 256 bytes.

`DitsGetMsgLength()` is useful for any message to be sent with `DitsObey()`, `DitsKick()`, `DitsSetParam()`, `DitsGetParam()`, `DitsInitiateMessage()`, `DitsTrigger()` and for arguments to be returned using `DitsPutArgument()`. (Note that higher level routines in DUI, DUL and GIT provide wraparounds of some of these for various cases) This leaves messages to be sent with `MsgOut()`, ERS and parameters being monitored.

Messages sent using `MsgOut()` have a fixed length, which in the current version of DRAMA (1.2) is 580 bytes. For parameters which are to be monitored by other tasks, you can run `DitsGetMsgLength()` on a parameter by using `SdpGetSds()` to access the SDS id of the parameter. The parameter values are sent in TAP (reply) messages.

ERS messages are the hardest ones to sort out. This is because error reporting occurs when things go wrong and sometimes things can go badly wrong, with different levels of the system reporting multiple error messages and recursion occurring. A single ERS message can have up to 30 error lines, each of 200 characters. A single message is sent when a call to `ErsFlush()`/`ErsOut()` is made and may also be sent when an action stage returns. In some cases, if an application is trying to recover from an error, it may send multiple ERS messages. In the current version of DRAMA (1.2) an ERS message with a one-line report requires 660 bytes. A full 30 line report takes 6692 bytes. This works out at 208 bytes for each extra line. So if you want to catch a full ERS report, you will want a reply buffer of about 7000 bytes. I often use 8000 bytes.

Buffer Sizes in Practice

So how does one actually work out buffer sizes in practice. Often, you make a stab and experiment. For a simple path where you are sending a GSOK message and expecting replies, I normally assume 1000 bytes is plenty to send the message and I am unlikely to be sending more then one message. So, MessageBytes is set to 1000 and MaxMessages to 1. This amount of space and some overhead will be allocated from the global buffer space of the target task. For replies, my first stab is to allow sufficient space for ERS messages, which is normally plenty for the other messages. This space will be allocated from the global buffer space of the task calling `DitsPathGet()`.

If you are sending large argument structures, say images, you may have to use `DitsGetMsgLength()` to determine the size of the buffer you need. Remember to consider both directions.

The next thing to consider is if there are likely to be bursts of messages which a task cannot process quickly enough. In this case, you have to increase the MaxMessages or MaxReplies arguments appropriately. This is often done by experimentation with some extra added just to be sure. Things to consider here are networking speeds (the network may make a task look slow, causing the IMP transmitter task's buffer to fill up) and the time it takes for user interfaces to format messages. The later is often a problem with bursts of error or logging messages - with user interfaces often having trouble outputting the messages as quickly as they come in.

If you are quite restricted in the amount of memory available, then you may want to use the above information to set the buffers sizes more precisely. You can make use of the `DitsGetPathSize()` routine to obtain the actual buffer sizes for a given path. If needed, the routine `DitsGetParentPath()` allows an action to get the path to the parent task, for use with `DitsGetPathSize()`.

To determine the global buffer space allocation, add up all the buffers that are to be allocated to that task and then add the overhead of 4000 Bytes. This overhead is used for the "self path" (default of 2000 bytes) and storing assorted information which must be accessed by other tasks. If often use 20000 as a starting point. This ensures there is sufficient space for multiple connections to the task, often useful when debugging.

Example calculation of buffer sizes.

As an example, let us consider my TICKER/TOCKER tasks. These tasks are used when testing DRAMA. TICKER is started first. TOCKER is then started and gets a path to TICKER. It then sends an Obey message to TICKER and waits for the response. It repeats this obey message a given number of times, although always waiting for the previous message to complete.

First consider the TICKER task, a simple task. It only needs space for one buffer from the TOCKER task, which will be used to send a single Obey message. Since this Obey message has no arguments, only 304 bytes is needed. I will choose 400 Bytes as the value for MessageBytes and one as the value for MaxMessages. Assuming only one task is going to talk to TICKER, this means TICKER has a global buffer space requirement of 400 bytes plus the overhead mentioned above (4000 bytes).

For messages from TICKER back to TOCKER, it is sufficient to allow space for the maximum ERS message. This gives a ReplyBytes value of 8000 and a MaxReplies value of 1. The TOCKER global buffer space requirement is therefore 12000 bytes.

Buffer space allocation   problems.

Now the above scheme generally works well, avoiding any delay in the sending tasks whilst waiting for the receiving tasks to read messages. But there are three problems you can hit, and when you hit them, it is often hard to work out what exactly has gone wrong. The problems are

1. There may be insufficient space available in the global buffer space when you try to allocate space using `DitsGetPath()`.
2. There may be insufficient space available in a buffer for you to ever send a particular message.
3. There may be a temporary lack of space due to the target task being slower then the sending task.

Each problem may occur in a number of cases. For example, problem 1 may occur due to task B running out of global buffer space, or task A running out of global buffer space or if the tasks are on remote networks, if either of the transmitter tasks involved run out of global buffer space.

Unfortunately, some errors are known about immediately by the task which requested the operation whilst others require an error message to be sent back to that task by another task in the system. Because of this, DRAMA is not yet consistent in handling these errors. Error codes tend to differ for what is basically the same error and there are a

couple of mistakes in the messages you get. We hope to fix this over time. The messages shown below are valid as of DRAMA version 1.2, but may change (to provide better information and more consistency) in the next few versions of DRAMA.

We now examine each case and the error messages you may receive. In each case, the target task (equivalent of task B in figure 19.3) is named TICKER . The initiating task (equivalent of A) is the DTCL user interface. Many of the error reports contain a number of lines, generally output by the ERS package. But normally there is one occurrence of an error code, such as DITS-F-NOSPACE, representing status code DITS__NOSPACE. This is the value the status variable will be set to, as formatted by the DTCL user interface's default error handler. In these examples. the ERS error reports are output but remember you can intercept these using the features provided by the ERS package, such as ErsPush() and ErsPop().

Running out of Global Buffer Space

Please note that all the examples here are generated by calls to DitsPathGet() (or routines which are wrap arounds of DitsPathGet()). A failure in DitsPathGet() means you can't send a message to the task. If you have already been sending messages to a task - ie. you have a valid path, you should look at the next section.

If a local task you are trying to connect to (eg. task B in figure 19.3) has insufficient space for your path, you will receive the following error sequence

```
##DTCL: Not enough space for new ring buffer of 50000 bytes
# DTCL:Task "TICKER" does not have sufficient global buffer space
left for our   connection
# DTCL:Failed to find path to task "TICKER".
# DTCL:obey:%IMP-E-NO_SPACE, Insufficient space left in buffer
```

Notice the clear mention of "global buffer space". Also note the mention of "ring buffer". This is the type of IMP buffer being allocated. The "ring buffer" message is generated internally by IMP where as the other messages were generated by DITS.

Things are a bit different when the target task is on a remote machine. The problem is that the local task does not have sufficient information to give as clean an error message. The message you get is

```
##DTCL:Failed  to  find  task  "TICKER@AAOSSI"-  %IMP-E-NO_SPACE,
Insufficient
space left in buffer.
```

But, on the remote machine, if the standard output of the "Receiver" task on the target task's machine is directed somewhere you can see it, you will also see the following appear at the same time.

```
# IMP_Receiver: Not enough space for new ring buffer of 30032 bytes
```

In both the above examples, it is the "bytes" argument to the call to `DitsAppInit()` in Task B which must be modified.

The third possibility, also in the remote task B case, is that is the transmitter task on the machine on which task A is running which is lacking space. This is very similar to the first case. The error message will be something like this

```
##DTCL: Not enough space for new ring buffer of 20032 bytes
# DTCL:Task "ImpTransmitter" does not have sufficient global buffer
space left for our connection to task "TICKER"
# DTCL:Failed to find task "TICKER@AAOSSI"- %IMP-E-NO_SPACE,
Insufficient
 space left in buffer.
```

You can again see the clear mention of "global buffer space" but there is a clear indication that it is the transmitter task, known as "ImpTransmitter" which has the problem. Here, to solve the problem, you must set the environment variable `IMP_NET_KBYTES` on the machine on which task A is running. You must do this before starting the networking, ie, before invoking `dits_netstart`.

Alternatively, the target task (Task B in figure 19.3) may have sufficient space, but the originating task (Task A in figure 19.3) may not have sufficient space for the reply buffers. In this case, there is a simple error message reported back to the originating task and some more information is printed to `stderr` by the target task. The report back to task A will be like this

```
#DTCL:Failed to find task "TICKER"- %DITS-F-CON_REJECTED,
Connection
by rejected by target task.
```

Unfortunately, this is not telling you much, as the originating task does not know anything more then that the connection was rejected — the `DIT__CON_REJECTED` error code could also be generated in other cases, such as the target task explicitly rejecting the connection (see `DitsPutConnectHandler()`). So you really need the `stderr` report from the target task. This looks something like this

```
##TICKER: Not enough space for new ring buffer of 83200 bytes
# TICKER:Failed to accept connection from task "DTCL",
%IMP-E-NO_SPACE, Insufficient space left in buffer
# TICKER:Task "DTCL" does not have sufficient global buffer space
left
 for our connection
```

Note that is was the target task, task TICKER, which reported the message. Again, "global buffer space" is mentioned in the error message.

As per the previous case, there are the two networking cases. First, when the target task is on a remote machine (again it is the originating task which does not have the space). The following message is returned by the originating task.

```
##DTCL:Failed to find task "TICKER@AAOSSI"-
%IMP-E-NO_LOCAL_SPACE, No space in originating task.
```

Here it is quite clear where the failure is, but no further information is available. But, if you have access to the `stderr` device of the "receiver" task on the originating task's machine, you will see something like this.

```
# IMP_Receiver: Not enough space for new ring buffer of 16640 bytes
```

The combination of the two messages is normally enough to work out what is happening. In both the above cases, it is the bytes argument to `DitsAppInit()` in Task B which must be modified.

This leaves one more case to consider, where the task that has insufficient space is the transmitter task on the remote machine. This is probably the most confusing case as it is the target task which has the information. The originating task will receive the following error.

```
##DTCL:Failed to find task "TICKER@AAOSSI"-
%DITS-F-CON_REJECTED, Connection by rejected by target task.
```

But if you have access to `stderr` for the target task, it will give a report like this

```
##TICKER: Not enough space for new ring buffer of 8320 bytes
# TICKER:Failed to accept connection from task "DTCL", %IMP-E-
NO_SPACE, Insufficient space left in buffer
# TICKER:Task "DTCL" does not have sufficient global buffer space
left for our
 connection
```

This message has a clear flaw. Whilst is mentions "global buffer space", it incorrectly says the buffer in question is in DTCL. In fact, it is in the local transmitter task. DRAMA has no way at present from determining where the task is. We hope to fix this in a later release. In this case, to solve the problem, you must set the environment variable `IMP_NET_KBYTES` on the machine on which task B is running. You must do this before starting the networking, i.e., before invoking "`dits_netstart`".

Messages which won't fit in a buffer.

Here, it is assumed that `DitsPathGet()` has succeeded and you have a valid path. We wish to consider the error messages you will receive if you attempt to send a message which won't fit in a buffer, even when it is empty. This error is always known about immediately by the sending task. There are two cases, the case where a task is initiating a message transaction (say by calling `DitsObey()`) and the case where a reply is

involved (`MsgOut()`, ERS messages, `DitsTrigger()` messages and action completion messages.)

In the first case, you will receive an error message at the initiating task something like this

```
##DTCL:764 byte message (and 32 byte header) cannot fit in 432 byte
buffer
# DTCL:Message won't fit in receiver's (TICKER) buffer, length is
746.
# DTCL:Failed to start Action "TICK" to "TICKER" task -
%DITS-F-NOSPACE, There is not sufficient space in the message
receiver for
this message.
```

The `DITS__NOSPACE` message is the significant code. You can see full details about an attempt to send a 764 byte message, a 32 byte header to a buffer which is only 432 bytes long.

Reply messages become a touch harder since there are some cases where you really want something to get through and other cases where the target task should just return a bad status. The most important case is action completion. Here, if the action completion message does not get through in some form, many systems will hang. What DRAMA does is to ensure there is always space in a reply buffer for a reply message stripped of any argument (set up using `DitsPutArgument()`). If it then can't fit the full message, it can send a message without it's argument and with a bad status value. In this case, the task which sent the original obey will get an error message like this

```
##DTCL:Action "TICK" to task "TICKER", completed with status =
%DITS-F-COMPSENDERR, Error sending completion message, sent without
the
 argument.
```

If the message in question is a `DitsTrigger()` message, then the normal error reporting scheme can be used. For example, we will get something like the following error at the originating task.

```
##DTCL:TICKER:1216 byte message (and 32 byte header) cannot fit in
1200 byte
 buffer
# DTCL:TICKER:Error sending reply (tap) message of size 1216 to
task DTCL - %IMP-E-CANT_FIT, Message is too large for this
connection
##DTCL:Action "TICK" to task "TICKER", completed with status =
%IMP-E-CANT_FIT, Message is too large for this connection.
```

`MsgOut()` will produce a similar report, although this it is rarely seen as most reply buffers are sufficiently long for a `MsgOut()` message.

If the message which can't fit through is an ERS message, then there is little DRAMA can do but dump most of the information to `stderr` in the target task of the original message (task B in this case.)   There are two ERS cases. First, if the failure occurred whilst the target task was calling `ErsFlush()`. In this case, the `ErsFlush()` status argument will be set to " `IMP__CANT_FIT`", Message is too large for this connection" and this value is often returned to the originating task. In the second case, where the flushing of error messages is occurring as part of an action stage completion, status will not normally be set bad.   In both cases, the `stderr` device of the target task (Task B) will receive a dump of the original error as well as details about the ERS failure. For example,

```
!!Error  Report  While  Flushing:2092  byte  message  (and  32  byte
header) cannot
 fit in 432 byte buffer
!!Error Report While Flushing:Error sending reply (tap) message of
size 2092
to  task  DTCL  -  %IMP-E-CANT_FIT,  Message  is  too  large  for  this
connection
TICKER\Dits___ErrOut: error outputting message
##TICKER:Error message 1, the quick brown fox jumps over the lazy
dog
# TICKER:Error message 2, the quick brown fox jumps over the lazy
dog
# TICKER:Error message 3, the quick brown fox jumps over the lazy
dog
# TICKER:Error message 4, the quick brown fox jumps over the lazy
dog
# TICKER:Error message 5, the quick brown fox jumps over the lazy
dog
# TICKER:Error message 6, the quick brown fox jumps over the lazy
dog
# TICKER:Error message 7, the quick brown fox jumps over the lazy
dog
# TICKER:Error message 8, the quick brown fox jumps over the lazy
dog
# TICKER:Error message 9, the quick brown fox jumps over the lazy
dog
ErsFlush:Message output error f358152
!!Error message 1, the quick brown fox jumps over the lazy dog
!!Error message 2, the quick brown fox jumps over the lazy dog
!!Error message 3, the quick brown fox jumps over the lazy dog
!!Error message 4, the quick brown fox jumps over the lazy dog
!!Error message 5, the quick brown fox jumps over the lazy dog
!!Error message 6, the quick brown fox jumps over the lazy dog
!!Error message 7, the quick brown fox jumps over the lazy dog
!!Error message 8, the quick brown fox jumps over the lazy dog
!!Error message 9, the quick brown fox jumps over the lazy dog
```

The double output of the message above is a bit annoying any may be removed at some stage. But you can see the details of the failure are output is the original message.

Buffer full errors

We now examine another class of errors. This is where `DitsPathGet()` has succeeded and the message will fit in the buffer if it were empty, but the buffer is not empty and as a result, a message won't fit. This type of problem normally occurs in one of two cases. The first is if the target task of the message in question is hung. In this case, there is nothing much that can be done to resolve the problem. Your system is in a mess. The more common case is the target task has been flooded with messages that it can't process in quick enough. This is sometimes a momentary problem and sometimes not. If the target task can never process messages quickly enough, you have a system design problem. This leaves us with the momentarily not quick enough case. It is this case that we have most chance of doing something about easily, although the messages below refer to all cases.

For these momentarily not quick enough cases, there are two possible solutions - expand the buffer size or wait until it is empty. Which solution you use depends on the particular circumstances involved. In some cases DRAMA will use the later technique automatically. This is done for action completion and ERS messages. Here DRAMA uses the "Request Buffer Empty Notification" facilities provided by IMP to allow it to send the message when the buffer eventually empties. See `DitsRequestNotify()` for more information.   Note the DITS connection routines (`DitsPutConnectionHandler()`) can disable this feature as can the flag `DITS_M_NO_FC_AC` to `DitsAppInit()`.

 What we will do here is look at the types of errors messages and where they indicate the problem is.

The first example concerns sending a message to a local task. Again the actual tasks being used are DTCL as the sender (equivalent of A in figure 19.3) and TICKER as the target (equivalent of B). In this case you will receive an immediate error from the call to `DitsInitiateMessage()` (or `DitsObey()` or `DitsKick()` etc.). This error will look something like this

```
##DTCL: Unable to find space for 304 byte message (and 32 byte
header)
# DTCL: Ring index 3, write count 3 read count 2
# DTCL: Write offset 336, read offset 0, length 688. Write is by
pointer.
# DTCL: Tested write offset 336, read offset 0.
# DTCL: Buffer has 104 bytes reserved for system use.
# DTCL:No imp space in receiver's (TICKER) buffer, length is 304.
# DTCL:Failed to start Action "TICK" to "TICKER" task.
# DTCL:obey:%DITS-F-NOSPACE, There is not sufficient space in the
message receiver for this message
```
❶
❷

❸

❹

You can see at ❶ what size message is being written. At ❷ you can see three items relating to the management of the buffer. The "write count" and "read count" are of most interest. The difference between them indicates how many unread messages there are in the buffer. In this case, there is one unread message. This value should give you some idea of how much slower the receiver is running compared to the sender. Note that the

available length in this buffer is calculated by subtracting from the "length" at ❸ the reserved system space at ❹. In this example the value 584 bytes. So assuming the TICKER task is running momentarily slow, you can probably fix this problem by adding roughly 300 bytes to the `MessageBytes` item when getting the path to TICKER. Alternatively, bump the `MaxMessages` item up by one.

The next case it consider is where task B is remote and it is still task B which has run out of space. Remember that in this case, it is the receiver task on task B's machine that is actually writing into task B. Since DRAMA can't work out what is going on immediately, it must report in the form of an error message being returned. In this case, task A gets a message with `DitsGetEntReason()` returning `DITS_REA_MESREJECTED`. The error code will be `IMP__NOSPACE` (which will probably be changed to `DITS__NOSPACE` at some stage in the future). For example.

```
##DTCL:Action "TICK" to task TICKER@AAOSSI rejected, reason =
"%IMP-E-NO_SPACE, Insufficient space left in buffer"
```

If you have access to the `stderr` device of the receiver task, you should see something like this

```
# IMP_Receiver: Unable to find space for 304 byte message (and 32
byte header)
# IMP_Receiver: Ring index 3, write count 3 read count 2
# IMP_Receiver: Write offset 336, read offset 0, length 688. Write
is by pointer.
# IMP_Receiver: Tested write offset 336, read offset 0.
# IMP_Receiver: Buffer has 104 bytes reserved for system use.
```

This gives useful information but unfortunately does not mention the tasks involved. You have to check for this output immediately the problem occurs otherwise it may be confusing.

The third case to consider in this set is again where the target task is remote but where it is the network tasks which are running slowly. It is that buffer to the transmitter task being used for communication with task B that is full. Here the message is exactly the same as for the local case. In the future, this may be changed to indicate it is the buffer in transmitter which is full.

There is one other forward message case. This is where the task is remote but the path is flow controlled. This is set by the `DITS_M_FLOW_CONTROL` flag to `DitsPathGet()`. When a path is flow controlled, the IMP sending routines return bad status if IMP is unclear if there is space at the receiving end for a message. Flow control allows you to ensure your message will get through. IMP uses a series of checkpoint messages to work out when the buffer is empty. You will see a message like this.

```
# DTCL:No imp space in receiver's (TICKER) buffer, length is 304,
Sync Needed
# DTCL:Failed to start Action "TICK" to "TICKER" task.
```

```
# DTCL:obey:%DITS-F-NOSPACE, There is not sufficient space in the
message receiver for this message
```

This is very similar to the local task buffer full case, but note the "Sync Needed" string appended to the first line. Also there is less low level information reported. To get around this problem, you can expand you buffer sizes (MessageBytes or MaxMessages) or you can use the `DitsRequestNotify()` routine. This can be used in any of the above cases, but is mandatory in this case. If just arranges for your action to be notified with a message when the buffer is ready to use again.

Buffer full for replies.

We no address the reply case. DRAMA normally uses flow controlled buffers (see above) for the reply direction. This allows DRAMA to make use of the IMP buffer empty notifications to ensure messages are eventually sent. This is done for ERS messages and action completion messages and should ensure that any messages that can be sent will be. The cost is possible delays in sending messages due to having to wait for the buffers to empt.

However, this is not done for `MsgOut()` and `DitsTrigger()` messages as a failure is normally not critical with these. As a result, you must expect failure due to buffer overflow from these calls. You can through explicitly use `DitsRequestNotify()` yourself to handle these cases. When the problem occurs, you should expect to get the `DITS__NOSPACE` error code returned by the routine. As per the previous examples, you can expect a bunch of ERS messages, similar to the above examples, which will help you determine where the problem was. If you want to handle the error, remember to use ERS facilities (such as `ErsPush()` and `ErsPop()`) to catch the ERS messages.

One problem with this facility is that once `DitsRequestNotify()` has been invoked on a path, you cannot send a message along that path until the notification has been received. Once again, DRAMA handles this internally for ERS and action completion messages and again for `MsgOut()` and `DitsTrigger()` you must handle it yourself if you desire the effect. When this happens, different error codes are returned. For `MsgOut()`, `DITS__NOTIFYWAIT_MSG` is returned. For `DitsTrigger()`, `DITS__NOTIFYWAIT_TRIG` is returned. Note that for each of the three possible errors here, the solution is to use `DitsRequestNotify()`. The reason for the different codes is so then if the errors are not caught, there is a clear indication of what went wrong.

Remember here that if `DitsRequestNotify()` is being required a lot, it is possible you should expand the reply buffer sizes (MaxReplies or ReplyBytes).

Part 3 – DRAMA User interfaces, using and development.

20.The Simple Standard Tools

Overview

21.DTCL

22.DJAVA

23.Developing User Interfaces – C API

Part 4 – Building, Running and releasing DRAMA programs.

24.DRAMA Directory Structures

25.Unix

26.VxWorks (Under Unix)

27.Generating DRAMA Makefiles

28.VMS

29. Microsoft Windows

30.Releasing DRAMA software

Part 5 – Other DRAMA Features.

31.IMP Startup Files.

32.Communicating with ADAM Tasks

Part 6 - DRAMA's documentation, building and installing DRAMA.

33.Documentation

34.Using the DRAMA CD ROM

35.Acquiring and/or building DRAMA for Unix/VxWorks

36.Acquiring and/or building DRAMA for VMS

37.Acquiring and/or building DRAMA for Microsoft Windows

Index

Bibliography

[i] John K. Ousterhout, Tcl and the Tk Toolkit (Reading, Massachusetts: Addison-Wesley). 1994

[ii] Jeremy Bailey, Self-defining Data System (SDS) (Anglo-Australian Observatory, Epping N.S.W, Australia, DRAMA Document number 7). 31st October 1994

[iii] Tony Farrell, Generic Instrumentation Tasking Specification (GIT) (Anglo-Australian Observatory, Epping N.S.W, Australia, DRAMA Document number 9). 18th January 1994

[iv] Tony Farrell, Distributed Instrumentation Tasking System (DITS) (Anglo-Australian Observatory, Epping N.S.W, Australia, DRAMA Document number 5). 22nd December 1998

[v] Tony Farrell, A portable message code system (MESS) (Anglo-Australian Observatory, Epping N.S.W, Australia, DRAMA Document number 6). 23nd February 1995

[vi] Tony Farrell, DRAMA Error Reporing System (ERS) (Anglo-Australian Observatory, Epping N.S.W, Australia, DRAMA Document number 4). 16th March 1994

[vii] Tony Farrell, Distributed Instrumentation Tasking System (DITS) (Anglo-Australian Observatory, Epping N.S.W, Australia, DRAMA Document number 5). 22nd December 1998

[viii] Tony Farrell, Distributed Instrumentation Tasking System (DITS) (Anglo-Australian Observatory, Epping N.S.W, Australia, DRAMA Document number 5). 22nd December 1998