ANGLO-AUSTRALIAN OBSERVATORY AAO/DRAMA_GUIDE_3
DRAMA Software Report 3
Version 0.4

Tony Farrell
05-Aug-93

Guide to Writing Drama Tasks

Contents

1 Introduction **3**
1.1 The Tasking Model 3

2 The Packages Available **5**

3 Writing basic DRAMA Tasks **6**
3.1 Status convention 7
3.2 Making Coffee 7
3.2.1 Getting started 7
3.2.2 The Exit Action Routine 9
3.2.3 The CoffeeMochaStage1 Action Routine 9
3.2.4 The CoffeeMochaStage2 routine 11
3.3 Making Tea 11
3.3.1 Setting Up 11
3.3.2 The TeaLapSang action routine 11
3.3.3 Kick routines 14
3.4 Control Tasks 15
3.4.1 Overview of CTEST 15
3.4.2 Getting Paths 15
3.4.3 Responding to Path Found messages 17
3.4.4 Starting actions in other tasks 19
3.4.5 Receiving the resulting messages 20
3.4.6 Exiting 21
3.5 User Interface Tasks 22
3.5.1 The Dui routines 23
3.5.2 The Ditscmd program 23
3.5.3 X-Windows interface 26
3.5.4 Other Input Sources 32

4 Parameter systems **32**
4.1 The Simple Dits Parameter System 33

5 Object Orientated techniques **34**
5.1 The Simple Spectrograph Task 34
5.2 Activation routines 36

6 Staging Library Routines **37**

7 Other facilities available **41**
7.1 Action context 41
7.2 User and Action Data Routines 41
7.3 DitsInitiateMessage 42
7.4 Uface timers 42

8 Include Files **42**

9 Compiling, Linking and Running **42**

10 Compatibilty with Starlink-ADAM **43**

11 Software Organisation **43**

A Example code **45**
A.1 Coffee.c 45
A.2 Tea.c 45
A.3 CTest.c 45
A.4 Ditscmd.c 45
A.5 Object Oriented programming examples 45

Revisions:

V0.0 03-Aug-1993 Update for new versions of [1] and [8]. Introduction of [9] and use of **StatusType**. Introduce section on Staging procedures. Delete section on Software Organisation to be replaced at some later stage by the unix equivalent.

1 Introduction

This document describes how to write programs using the **DRAMA** software environment. **DRAMA** programs, known as Tasks, are normally used to control complex and distributed instrumentation systems in a soft real time [1] environment, spread across machines of different architectures and operating systems. In addition **DRAMA** could be used to implement any distributed application in such an environment.

This document does not attempt to describe in detail the various packages which make up **DRAMA**. It shows instead how these package are put together to produce a working system. Each package is fully described an appropriate document.

1.1 The Tasking Model

A *TASK* is a software object which responds to and initiates messages. The object is normally implemented as a separate process within a multiple process operating system such as **VMS**, **UNIX** or **VxWorks**.

The *Fixed Part* is that part of a task common to all tasks. It provides the message system interface, parameter system and controls action scheduling. The *Application Part* is that part provided by the programmer to implement his specific application.

A **Dits** task is similar to a Starlink[2] ADAM instrumentation task in its design. This style of task design has been driven by several different requirements-

- A task responds to messages with/without arguments which 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 routine 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 incomming 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

[1] The networking side of **DRAMA** may make it unsuitable for *hard* real time systems, i.e. systems where there is considerable cost if a system does not respond on time. It may be possible to port the networking layer to such a system, but the authors have not considered all the implications of such requirements

[2] Starlink is a project which provides computing support to U.K. Astronomy Community.

of multi-threading although the threads must control there 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** and **VxWorks**. As long an 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 about messages, from other tasks.

The above points results in a task with the following basic structure

```
loop until exit message received
    read next message
    initiate response to message
    if not complete
         then arrange a message to initiate the next stage (reschedule)
    else
         send completion message.
end loop
```

Of course this is simplistic. For example if the response to a message is CPU intensive it may not be efficient to reschedule frequently. To support this, we supply the ability to determine if a message is queued. A CPU intensive task can then reschedule only when it knows a message is available.

Experience has led us to determine several events which may be required to cause a reschedule

- Simple action staging. This is when you simply wish to break a CPU intensive activity into several parts to ensure the task can respond to other messages. When an action `stages` a message is immediately queued to reschedule that action, which will be done after any other messages already in the queue have been processed.
- Reschedule after a timer expiry. Similar to staging, but the reschedule message is not queued for a certain period of time. (A stage just becomes a reschedule with a delay of 0). Such time expiry reschedules can be used to implement timeouts if required.

- Interrupt triggered reschedules. These may or may not be pure hardware interrupts. Under **VAX/VMS** you probably want an `AST` (Asynchronous System Trap) to trigger a reschedule, not the actual hardware interrupt. Under **UNIX** the equivalent is a software signal.
- Reschedules triggered by the reception of messages from subsidiary tasks. A task can send messages to other tasks (called subsidiary tasks). When such a message completes it is desirable for the initiator to be told by rescheduling.

2 The Packages Available

The **DRAMA** environment implements the above tasking model in a portable way. It is written entirely in C, but is optimized for each of the systems on which it runs. It currently only supports systems which use the internet protocol for networking, but if required could be ported to any other networking system.

DRAMA consists of the following packages-

DITS - The *Distributed Instrumentation Tasking System* is the core of the **DRAMA** software environment. **Dits** enforces the basic structure of the task and interfaces it to the messages system. **Dits** is described in [1].

IMP - The *Interprocess Messages Passing System* provides low level message passing primitives. **Imp** is optimized to use the fastest possible message passing technique between tasks running on the one machine. Additionally it transparently allows messages to be sent to tasks on any other machine accessible via the internet network. It is described in [2].

SDS - The *Self-defining Data System* is a system which allows the creation of self-defining hierarchical data structures in a form which allows data to be moved between different machine architectures. It is described in [3].

ARG - The *Argument Package* is a simple package supplied as part of **SDS**. It provides simple techniques for the building and interpreting of action arguments. See [3] for more details.

ERS - The *Error Reporting System* provides facilities for constructing and storing reported error messages and for the delivery of those messages to the user via a technique appropriate to the program being run. The **Ers** routines are described in [4].

MESS - The *Portable Message Code System* provides a utility program (MESSGEN) which generates unique integer message codes. The **Mess** routines allow text to be associated with each message code. The input file and message codes generated are compatible with the **VMS**message utility. See [5] for more details.

SDP - The *Simple Dits Parameter System* is provided as part of **Dits**. It provides a simple parameter system for dits tasks. See [1] for more details.

DMON - The **Dmon** routines provide a technique which allows remote monitoring of a task's parameters using an AAO Adam Utask interface running on a VAX. See [1] and [6] for more details.

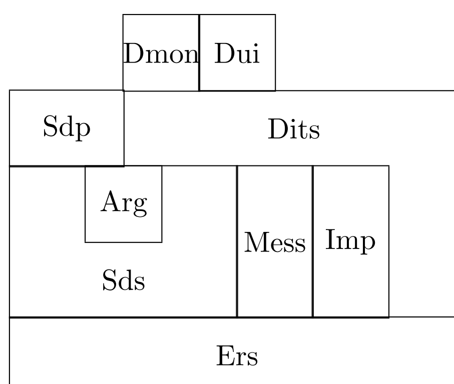

Figure 1: **DRAMA** layering diagram

DUI - The **Dui** routines provide techniques which assist the implementation of User interfaces. See [1] for more details.

The **Imp** routines should not be used directly by application code as this may cause problems with their use by **Dits**.

Figure 1 shows the layering of **DRAMA**.

3 Writing basic DRAMA Tasks

So how do you write a **DRAMA** task and what can it do? There are three broad classes of **DRAMA** tasks. The most basic simply respond to external messages. They may for example control an RS232 port, accepting messages which are simply sent to the RS232 port, with the response being sent to the initiator of the message.

The next level up is the Control Task. This task may control one or more other **DRAMA** tasks. Other than the job they do there is really little difference between the basic task and Control task, although Control tasks are likely to be much more complex.

The final level is the User Interface Task. The major difference is that this task type is the one to get things going. The basic and control task types only do something in response to an outside message. As a result they always have somewhere to send the results of messages (such as error reports). This is not true for User Interface Tasks. These will often send messages to other tasks as a result of non message system events - such as the user typing a command. Therefore user interface tasks must themselves handle getting the results back the to user.

In practice there is nothing to stop a single task performing all of the above functions although the result may be clumsy.

We start off by looking at two examples of basic tasks, then follow up with a control task and a simple user interface task.

3.1 Status convention

All routines provided by the tasking system will follow the Starlink status convention, i.e. each routine will have as its last argument a status variable. Normally, if this status is not equal to 0 (`STATUS__OK`) the routine will return immediately. The routine can set the status to a non zero value when it encounters an error. The error code should be a globally unique value calculated using the equation

$$CODE = 134250498 + 65536 \times < fac > +8 \times < mes >$$

Here, $< mes >$ is the message number (in the range 1 to 4095) assigned to the error condition by the author of the subroutine library and $< fac >$ is the facility number (in the range 1 to 2047) allocated to this subroutine library by the ADAM Support Group. See [5] for details on how to generate such error codes.

Status varibables should be of type **StatusType**, defined in the `status.h` include file. **StatusType** will be the smallest integer type large enough to store the status code.

An exception to this convention is made for functions returning a single value which will always be valid. Such functions do not require a Status argument. Some error handling functions may also need to handle status differently to obtain the desired effects.

3.2 Making Coffee

The Coffee task is an very simple example of a **DRAMA** task. It responds to six different commands (also known as actions). The **EXIT** command causes the program to exit. All tasks should support the **EXIT** message. The **MOCHAn** commands (n = 1 to 5) output a couple of messages at timed intervals and exit. The full code for `coffee.c` can be found in appendix A.1.

3.2.1 Getting started

Normally, when a C program starts up the function **main()** is called by the C run time library. **Main()** is the main program of a C program. Unfortunately, not all systems use a **main()** routine. In systems such as **VxWorks** all programs must have unique entry points. The folowing bit of code handles this complexity. It is only compiled on systems which need a **main()**. It just transfers control to the **coffee()** routine which does the actual initialisation.

```
int coffee();
#ifdef DITS_MAIN_NEEDED
    extern int main()
    {
        return(coffee());
    }
#endif
```

To set up **Dits**, a **Dits** task would normally call

1. **DitsInit()** - To initialise dits.
2. **DitsPutActionHandlers()** - To associate action names with functions.
3. **DitsMainLoop()** - To handle incoming **Dits** messages.
4. **DitsStop()** - To shutdown dits.

In coffee, it looks like this:-

```
int coffee()
{
    StatusType status = STATUS__OK;

    DitsInit("COFFEE",BufSize,0,&status);
    DitsPutActionHandlers(ActionMapSize,ActionMap,&status);
    DitsMainLoop(&status);
    return(DitsStop("COFFEE",&status));
}
```

The Task Name - A task must have a name by which it is known to the message system. It is also used in error messages output by the task. In this case - "COFFEE". Note we supply to task name to both **DitsInit()** and **DitsStop()** since if **DitsInit()** failed earily in its sequence, the error output routines in **DitsStop()** may not have the taskname available.

The length of this name should be less than the constant `DITS_C_NAMELEN` (20 characters) including the null terminator.

ActionMap and ActionMapSize - **ActionMap** is an array which maps action names to routines. Action names are the names of the commands which the task is prepared to accept. For each action you must define -

- The Routine to be called when an Obey message is received. An Obey message is sent by one task to another to start the action in the second task.
- The Routine to be called when a Kick message is received. If an action is already active, a Kick message is used to influence the operation of the action, such as to request an abort.
- A user defined code. This code will be available to the action when it is run. Any long integer value can be specified. It could be used to differentiate actions which use the same routine.
- The name for the action. This can be up to `DITS_C_NAMELEN` characters (currently 20).

We must also define the length of the array (the number of actions).

In coffee we define them as follows-

```
DitsActionMapType ActionMap[] = {
     {CoffeeMochaStage1, 0, 1, "MOCHA1" },
     {CoffeeMochaStage1, 0, 2, "MOCHA2" },
     {CoffeeMochaStage1, 0, 3, "MOCHA3" },
     {CoffeeMochaStage1, 0, 4, "MOCHA4" },
     {CoffeeMochaStage1, 0, 5, "MOCHA5" },
     {CoffeeExit   , 0, 0, "EXIT" }
};
int ActionMapSize = sizeof(ActionMap)/sizeof(ActionMap[0]);
```

Note the **CoffeeMochaStage1()** and **CoffeeExit()** are routine names. We don't have Kick routines, indicating that none of these actions accept kick messages.

BufSize - This variable is used as the **bytes** argument to **DitsInit()**. To work out this value you should first determine the size required to allocate all the buffers to be used in messages sent to this task. These buffers are allocated by **DitsGetPath()**. This value should be increased by about 70% to allow for overheads. We look more into **DitsGetPath()** later. For a task like **COFFEE** an appropriate value is about 5000 Bytes.

```
int BufSize = 5000;
```

3.2.2 The Exit Action Routine

The routine **CoffeeExit()** is very simple, being nothing more than-

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

All this routine does is request that the task exit. We do not bother with the normal status check on entry since this is done by **DitsPutRequest()**. The exit status of the task is the value of status that the routine exits with, which in this case will always be ok.

3.2.3 The CoffeeMochaStage1 Action Routine

This routine will be called when a message for any of the **MOCHAn** actions is received. All it does is output a message and then reschedule.

```
static void CoffeeMochaStage1(StatusType *status)
{
    char name[DITS_C_NAMELEN];  /* For the name of the action */
    DitsDeltaTimeType delay;    /* To calculate the delay     */
    int seconds;
```

```
    if (!StatusOkP(status)) return;
/*
 *  First the the name of the action is output in a message
 */
    DitsGetName(sizeof(name),name,status);
    MsgOut(status,"Starting %s action",name);
/*
 *  Use the code for this action as the action delay (just for fun)
 */
    seconds = DitsGetCode();
    DitsDeltaTime(seconds,0,&delay);
/*
 *  Put the delay and request it
 */
    DitsPutDelay(&delay,status);
    DitsPutRequest(DITS_REQ_WAIT,status);
/*
 *  Use the Stage2 handler for the next stage of this action.
 */
    DitsPutHandler(CoffeeMochaStage2,status);
}
```

The macro function **StatusOkP()** will return true if its argument (a pointer to a **StatusType**) is equal to `STATUS__OK`. A similar macro function - **StatusOk()** - exists. Its argument is a plain **StatusType**, not a pointer.

The **DitsGetName()** function is used to get the actual name of the action, which is output to the user using **MsgOut()**. Note that the second and subsequent arguments to **MsgOut()** work the same way as the first and subsequent arguments to the C Run-Time library function **printf()**. You should not use any special characters, such as \n or \003, since the meaning of these characters is dependent on the user interface in use.

The routine **DitsDeltaTime()** converts a time in seconds and microseconds into a value which can be used in a call to **DitsPutDelay()**. The delay (in seconds) that we use is simply the number specified as the code for this action in the **DitsMap** array. We get this by calling **DitsGetCode()**. It could be any valid value.

After putting the action delay we want, we request that the action be rescheduled using **DitsPutRequest()** with a request of `DITS_REQ_WAIT`. While this request is outstanding additional obey messages for this action will be rejected.

The last call, to **DitsPutHandler()**, is used to set the name of the routine to be called on the next entry to this action. The entry will occur when the delay expires. We do not have to call **DitsPutHandler()**, in which case, the same routine will be called again and we could use the sequence number returned by **DitsGetSeq()** to differentiate each entry. This number is set to 0 when a new invocation of the action is started and is incremented by one for each reschedule of the action. **DitsPutHandler()** is a bit more efficient and may allow a more appropriate and

neater program structure. (The efficiency effect increases as the program structure gets more complex).

3.2.4 The CoffeeMochaStage2 routine

This routine is called as a result of a reschedule of an action. It just outputs a message and exits.

```
static void CoffeeMochaStage2(StatusType *status)
{
    char name[DITS_C_NAMELEN];
    if (!StatusOkP(status)) return;
    DitsGetName(DITS_C_NAMELEN,name,status);
    MsgOut(status,"Finishing %s action",name);
}
```

Note that no call to **DitsPutRequest()** is made. The default request is `DITS_REQ_END` which indicates the action is to complete and a completion message sent to the originator of the action. The status of the completion message is the value of status on exit from this action.

3.3 Making Tea

The **Tea** task is a more complex example. The source code can be found in appendix A.2.

3.3.1 Setting Up

In the tea Task, the ActionMap looks like this-

```
DitsActionMapType ActionMap[] = {
     {TeaLapSang, TeaLapSangKick, LAP1, "LAPSANG1" },
     {TeaLapSang, TeaLapSangKick, LAP2, "LAPSANG2" },
     {TeaLapSang, TeaLapSangKick, LAP3, "LAPSANG3" },
     {TeaLapSang, TeaLapSangKick, LAP4, "LAPSANG4" },
     {TeaLapSang, TeaLapSangKick, LAP5, "LAPSANG5" },
     {TeaExit   , 0, 0, "EXIT" }
};
```

In addition to the Obey routines (**TeaLapSang()**) we also have a Kick routine - **TeaLapSang-Kick()**. The names `LAP1` to `LAP5` are just macros which define the codes for each action.

3.3.2 The TeaLapSang action routine

The **TeaLapSang()** routine is called for all entries of all actions except when responding to Kick messages. In order to differentiate the actions, it uses the action code, fetched with -

```
long int code = DitsGetCode();
```

It uses the action sequence number to determine which entry of the action, using code like this-

```
long int seq = DitsGetSeq();
      ...
if (seq == 0)
{
    /* Do this on the first entry of an action */
}
else
{
    /* Do this on all other entries of an action */
}
```

The LAPSANG1 Action - The first entry of the **LAPSANG1** action first writes a message using **MsgOut()** (which is common to all the actions) and then reports an Error using **ErsRep()**. **ErsRep()** is similar to **MsgOut()**. The first argument is a flag mask indicating to the user interface how the message should be displayed. The possible values are

Flag	Meaning
ERS_M_NOFMT	Don't format the string. Any formating 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.

These flags are really only hints to the user interface. A simple user interface may ignore them.

The second argument is status, but this does not work in the normal way. **ErsRep()** will work regardless of the value of status and the value of status is included in the message sent to the user interface, where it may or may not be used.

The third and fourth arguments are again similar to the first and second arguments to the C Run-time library routine **printf()**.

After calling **ErsRep()**, **ErsFlush()** is called. This routine will cause any messages reported by **ErsRep()** to be flushed to the user interface. It is normally only called if code wishes to continue after an error but wishes reported error messages to be displayed. Use **ErsAnnul()** to cancel error messages. Both these routines will clear status. See [4] for more details.

The action then reschedules at an interval based on the action code. When the reschedule occurs (seq > 0) the action outputs a messages and completes.

The LAPSANG2 and LAPSANG5 Actions - These two actions simply output a message and reschedule once. The output another message and complete.

The LAPSANG3 action - This works the same as **LAPSANG2** except that it attempts to get the value of its argument. Action arguments are **SDS** items constructed by the task which sent the message. The lines-

```
DitsArgType argId = DitsGetArgument();
ArgGeti(argId,"COUNT",&count,status);
```

first fetches the **SDS** id of the argument to this action. It then tries to find the the value of an item in this structure named “COUNT”. The “i” in the name **ArgGeti()** indicates the value should be returned as an integer, being converted if necessary and possible.

We can also send arguments with a completion message. This is done by using the **ARG** routines to construct the argument and **DitsPutArgument()** to notify **Dits** of it. The argument is only sent if the action completes on this entry.

The next bit of code indicates how we handle errors -

```
if (!StatusOkP(status))
{
    char errmess[100];
    MessGetMsg(*status,0,sizeof(errmess),errmess);
    ErsOut(0,status,"LAPSANG3:Error getting argument, status = %s",errmess);
}
```

The routine **MessGetMsg()** is used to fetch any text which was associated with the status code. We then use **ErsOut()** to report the error. **ErsOut()** is the same as a call to **ErsRep()** followed by a call to **ErsFlush()**. We used it because we wish to continue the action regardless of this error. You could also use **DitsErrorText()** instead of **MessGetMsg()**. **DitsErrorText()** returns a pointer to the message, which is held in an interal buffer which is overwritten by subsequent calls. See [1] for details.

MessGetMsg() only knows about the error codes registered with **MessPutFacility()**. **Dits** registers the error codes for all the packages mentioned above, but you must explicitly register the error codes for any other facility used. See [5] for more details.

The LAPSANG4 action - On its first entry **LAPSANG4** works the same as **LAPSANG2**, but its second and subsequent entries are handled differently. This action will reschedule a number of times (depending on the value supplied as an argument to **LAPSANG3**) and each time it calls **DitsTrigger()** as follows-

```
if (seq <= 1)
    DitsDeltaTime(0,200,&delay);
DitsTrigger(0,status);
if (!StatusOkP(status))
{
    char errmess[100];
    MessGetMsg(*status,0,sizeof(errmess),errmess);
```

```
    ErsRep(0,status,"LAPSANG4:Failed to trigger control task, status = %s",
                                                                      errmess);
}
else
{
    DitsPutDelay(&delay,status);
    DitsPutRequest(DITS_REQ_WAIT,status);
}
```

The routine **DitsTrigger()** sends a message to the originating task of the message which started this action, causing the originating action to reschedule. The first argument to **DitsTrigger()** is an id of an **SDS** item to be sent as part of the message. The originator can retrieve this. By suppling a value of 0 as the argument we are indicating that there is no value to be sent.

DitsTrigger() actually works in a similar way to **MsgOut()** and **ErsFlush()**, but unlike the later two which are intended to send messages to the user, **DitsTrigger()** is intended to send messages to the originating action.

Another interesting point about this bit of code is that the call to **DitsDeltaTime()** is only made once. As the same value is used each time, we use a static variable and calculate the delay once only.

3.3.3 Kick routines

Kick routines are called in the context of action which is already active. They are normally used to cause the action to abort, but may have other uses, such as changing the way an action works.

TeaLapSangKick() does very little -

```
static void TeaLapSangKick(StatusType *status)
{
    long int code = DitsGetCode();
    DitsDeltaTimeType delay;
    if (!StatusOkP(status)) return;
    MsgOut(status,"Action LAPSANG%d Kicked",code);
    DitsDeltaTime((code/2),0,&delay);
    DitsPutDelay(&delay,status);
    DitsPutRequest(DITS_REQ_WAIT,status);
}
```

simply outputting a message and rescheduling. A Kick routine can do one of three things-

1. Reject or ignore the Kick. If a Kick routine does not call **DitsPutRequest()**, then the normal sequence of the action will not be changed. By setting status to something other than `STATUS__OK`, an error message is sent to the originator of the Kick message. If status is not set, then no message is sent to the originator.

2. Cause the action to abort. By using **DitsPutRequest()** to put a request of either `DITS_REQ_END` or `DITS_REQ_EXIT`, the action will terminate. In the later case the task will exit. The status returned by the Kick routine will be the exit status of the action.

3. Cause the action to reschedule in a different way. By using one of the other requests, such as `DITS_REQ_WAIT`, the Kick can cause the action to alter its rescheduling. This is useful if, when aborting an action, you must wait for hardware to abort correctly before the action can completes.

3.4 Control Tasks

Control tasks are used to coordinate a number number of different basic tasks. For example you may have a detector and spectrograph system which are independent at the physical level, but are to be used together. A control task may coordinate the operation of the detector system with the spectrograph, making sure the detector is not exposed in the middle of configuring the spectrograph.

This section looks at some of the techniques used to write control tasks. The program we look at, used to control the **Tea** and **Coffee** tasks, is listed in full in appendix A.3.

3.4.1 Overview of CTEST

This control task only supports two actions, the standard **EXIT** action and **BREW**, which does the real work. **BREW** does not support kick messages.

The **BREW** action is responsible for the following activities-

- Getting paths to the **Tea** and **Coffee** tasks.
- Obeying actions **MOCHAn** (n = 1 to 5) in task **Coffee**.
- Obeying actions **LAPSANGn** (n = 1 to 5) in task **Tea**.
- Responding to any messages sent by the **Tea** and **Coffee** tasks.
- Handling the completion messages of the actions started in **Tea** and **Coffee**.

3.4.2 Getting Paths

A `Path` is a two way communications link to another task. To be able to send a message to another task we must have a path to that task. We use the **DitsGetPath()** function to get a path to a task. The routine **CTestFindPaths()** contains the following calls-

```
DitsGetPath("TEA",MESSAGEBYTES,MAXMESSAGES,REPLYBYTES,MAXREPLIES,
                                    &teaPath,&teaTransid,status);
DitsGetPath("COFFEE",MESSAGEBYTES,MAXMESSAGES,REPLYBYTES,MAXREPLIES,
                                 &coffeePath,&coffeeTransid,status);
```

The first argument is the name of the task (**TEA** or **COFFEE**). This must be the name the task specified when it called **DitsInit()** If the task is on a remote machine use the format `task@node` where `node` is the internet address (by name or internet number) of the node on which the task is running.

The second argument specifies the maximum size in bytes of messages to be sent to that task, while the third is the number of such messages which may be sent before the target can get around to responding. If you multiply these two values and add a factor for overhead (say 20 to 30%), you will obtain the amount of space which will be allocated in the message buffer of the target task. The target must have allocated this much space for each task which will connect to it. This is done by the target when it calls **DitsInit()** I used a value of 400 for the size of a message. This is plenty for Obey or Kick messages with simple arguments containing one or two scaler values. Since five such messages are sent to the target in quick succession, `MAXMESSAGES` is set to five.

The fourth and fifth arguments are similar to the second and third, except that they indicate the space to be allocated in this task for messages sent from the target task to this task. This may include in addition to action completion messages, messages sent with **MsgOut()** and **ErsFlush()**. The later may be quite long so 800 seems an appropriate value for `REPLYBYTES` while experience suggests 12 as an appropriate value for the number of replies in this example.

At this stage you can work out appropriate values for the **Bytes** argument to **DitsInit()**. In the target tasks (**Tea** and **Coffee**) we need about 400 x 5 = 2000 Bytes + overhead. Thus in the order of 3000 Bytes are required. I went to 5000 just to be sure. In this task we need 800 x 12 x 2, plus about 1000 for the task which sends **BREW** command, about 20000. Just to be sure use 30000.

The sixth argument to **DitsGetPath()** is the address of a variable of type **DitsPathType**. If **DitsGetPath()** returns with status ok this address will contain the path. The path variable may or may not be valid at this point. If we have previously obtained a path to this task the path will be valid and can be used immediately to send messages to the target task. If we have not previously obtained a path to this task a network transaction will be required to set it up. The value returned in the path argument is the correct path, but cannot be used until this transaction has completed.

The seventh argument is the one concerned with this network traffic. Most **Dits** routines which initiate message transactions have an argument of this type. It is known as the "transaction id". You should supply the address of a variable of type **DitsTransIdType**. In the case of **DitsGetPath()**, if a value of zero is returned as the transaction id no network transaction is required to set up the path. If a non-zero path is required a network transaction is required. The action must reschedule to await its completion. As you may initiate many transactions in one entry you may need the transaction id to differentiate the various transactions.

So there are two cases after a successful call to **DitsGetPath()** depending on whether the transaction id is zero or non zero. In the later case we must reschedule to await the completion of the transaction. **CTest** handles it like this-

```
if ((teaTransid == 0)&&(coffeeTransid == 0))
{
```

```
    DitsPutHandler(CTestStartActions,status);
    DitsPutRequest(DITS_REQ_STAGE,status);
}
else
{
/*
 *  Put the handler for the next stage, which will occur when either
 *  the path is found OR the timeout expires, which ever comes first.
 */
    if (TIMEOUT > 0)
        DitsPutDelay(&timeout,status);
    DitsPutHandler(CTestPathFound,status);
    DitsPutRequest(DITS_REQ_MESSAGE,status);
}
```

If we have both paths immediately the task just continues. It does this by setting a new handler for the next entry of this action, requesting an action stage and then returning. We could have called **CTestStartActions()** directly but the above technique is cleaner and in a complex task would allow other messages to be handled.

In the second case we reschedule to await the completion of the transactions started by **DitsGetPath()**. We use **DitsPutDelay()** set up a timeout on getting the path.

3.4.3 Responding to Path Found messages

If we had to wait for the get path transaction to complete, then **CTestPathFound()** will be the next routine called. When it is called the routine **DitsGetReason()** will return the reason for the entry the status associated with the entry.

There are only three possible reasons at this stage-

1. The timeout set up above has been triggered. The reason code will be `DITS_REA_RESCHED`. Either we have underestimated the timeout or something is wrong. In either case we want to report the error and exit the **BREW** action. In order for the originator of the **BREW** action to know that something is wrong we should return a non-zero status. We could have created our own message facility (see [5]) but this is a bit complex for such a simple task. To help, **Dits** provides two codes for use by applications. `DITS__APP_TIMEOUT` is used in this case. The associated text indicates an application timeout. The other code available is `DITS__APP_ERROR` which indicates a non-specific application error. Always report an error using **ErsRep()** when you set status to one of these codes. The associated code in **CTest** is-

   ```
   if (reason == DITS_REA_RESCHED)
   {
       *status = DITS__APP_TIMEOUT;
       ErsRep(0,status,"Timeout trying to get paths");
   }
   ```

2. We were unable get a path to a task. This would normally only occur if a remote task does not exist, if an error occurs setting up the connection or if the target task dies or rejects the connection. If a local task does not exist then **DitsGetPath()** would have returned an error.

   The variable **reasonstat** returned by **DitsGetReason()** will contain a status code indicating a reason for the error. We may also want other information such as the name of the task which died, the transaction id of the outstanding transaction and the path to the task which died. We can use **DitsGetEntInfo()** to get this information. In **CTest** we handle it as follows-

```
else if (reason != DITS_REA_PATHFOUND)
{
    char errmess[100];
    char name[DITS_C_NAMELEN];
    DitsTransIdType transid;
    DitsPathType path;
    DitsGetEntInfo(sizeof(name),name,&path,&transid,&reason,
                                   &reasonstat,status);
    MessGetMsg(reasonstat,0,100,errmess);
    *status = reasonstat;
    ErsRep(0,status,"Failed to get path to task %s: %s",name,errmess);
}
```

   Note that the actual reason code at this point should be `DITS_REA_PATHFAILED`, but we use this bit of code as a catch-all just to be careful.

3. We have successfully got a path to a task. Again, **DitsGetEntInfo()** can be used to get the details. **CTest** handles is like this-

```
else /* reason = DITS_REA_PATHFOUND */
{
    char name[DITS_C_NAMELEN];
    DitsTransIdType transid;
    DitsPathType path;
    DitsGetEntInfo(sizeof(name),name,&path,&transid,&reason,
                                   &reasonstat,status);
    if (path == teaPath)
        teaTransid = 0;
    else
        coffeeTransid = 0;

    if ((teaTransid == 0)&&(coffeeTransid == 0))
    {
        DitsPutHandler(CTestStartActions,status);
        DitsPutRequest(DITS_REQ_STAGE,status);
    }
```

```
        else
        {
            if (TIMEOUT > 0)
                DitsPutDelay(&timeout,status);
            DitsPutRequest(DITS_REQ_MESSAGE,status);
        }
    }
```

If we now have both paths we just stage to **CTestStartActions()**. Otherwise we reset the timeout and wait for the next message.

3.4.4 Starting actions in other tasks

The **CTestStartAction()** routine actually starts the various actions in the **Tea** and **Coffee** tasks. The routine used to do this is **DitsObey()**. The call -

```
DitsObey(coffeePath,name,0,&transid,status);
```

starts an action in the coffee task. The first argument is the path to the coffee task, obtained using **DitsGetPath()**. The second argument is the name of the action to start, in this case one of **MOCHAn** where n = 1 to 5. The third argument to the call is the argument to the action. There is no argument here, so we specify zero. The fourth argument is the id of the resulting transaction.

Remember that the **LAPSANG3** action in the **Tea** task supports an argument. The following code is used to construct and send that argument-

```
DitsArgType id;
ArgNew(&id,status);
ArgPuti(id,"COUNT",TEAARG,status);
DitsObey(teaPath,name,id,&transid,status);
SdsDelete(id,status);
SdsFreeId(id,status);
```

The **ArgNew()** function creates a new argument returning the identifier to it. **ArgPut()** will create a new integer element named "COUNT" and put the value specified into it, assuming "COUNT" does not already exit. If "COUNT" already existed as a scaler item, the value is converted if necessary and put into that item. We then specify the id in the call to **DitsObey()**. If we are not going to use the argument id again, its a good idea to delete it. Since the argument id is nothing more then a **SDS** id, and **ARG** does not provide deletion routines, we use **SdsDelete()** and **SdsFree()** to tidy up.

So what do we do once we have started (subsidiary) actions in another (subsidiary) task? The subsidiary task can send various messages to our task in the context of subsidiary action. There are five types of messages which may be sent -

- Transaction completion messages - These indicate the subsidiary action has completed. After one of these is received there will be no more messages from the subsidiary action - the transaction is complete.
- Transaction failure messages - These indicate there was some error in starting the subsidiary action, such as the task does not support an action of that name. After one of these is received there will be no more messages from the subsidiary action - the transaction is complete.
- Trigger messages - These messages are sent by the subsidiary action calling **DitsTrigger()**. They allow complex communication between the parent and subsidiary actions.
- Informational Messages - These messages are sent by the subsidiary action when **MsgOut()** is called. They are normally forwarded automatically to the parent of the parent action and so forth, until they reach the user interface to the system. It is possible to intercept them if required.
- Error Messages - These messages are sent by the subsidiary action when **ErsFlush()** or **ErsOut()** are called. They are normally forwarded automatically the parent of the parent action and so forth, until they reach the user interface to the system. It is possible to intercept them if required.

To await these messages your task must reschedule in the same way that is done to wait for get path transactions to complete -

```
if (TIMEOUT > 0)
    DitsPutDelay(&timeout,status);
DitsPutRequest(DITS_REQ_MESSAGE,status);
DitsPutHandler(CTestResults,status);
```

The handler **CTestResults()** will be called whenever a message is received relating to the outstanding transactions. As mentioned above, some messages are handled automatically without your handler routine being called. You can change this using **DitsInterested()** and **DitsNotInterested()**.

It is considered inappropriate for a task to complete with outstanding transactions. Either you can keep track of them yourself, as **CTest** does, or the function **DitsCheckTransactions()** can be used to check for them.

3.4.5 Receiving the resulting messages

When your handler is called you can again use **DitsGetReason()** to get the reason for the entry. See the [1] for a list of all the possible reasons for an entry. The **CTest** task only expects the following -

1. The timeout has expired. See section 3.4.3 for more details.
2. A trigger message is received. This is caused by a subsidiary action calling **DitsTrigger()**.

3. An action has completed. We can use **DitsGetEntInfo()** to get the path and id of the transaction. **Reasonstat** will contain the exit status of the action.

4. An action was rejected. Again we can use **DitsGetEntInfo()** to get the path and transaction id of the transaction completing. **Reasonstat** will contain the status associated with the failure.

5. A task to which there is an outstanding transaction has died. There will be one entry for each outstanding transaction and those transactions will be considered complete when the entry returns. Again use **DitsGetEntInfo()** to get more information, including in this case, the name of the task which has died.

For all the other possibilities **CTestResult()** calls the routine **DitsPrintReason()**. This routine will use **ErsRep()** to output the details of the entry including, if appropriate, the associated status.

Once the first completion message has been received **CTest()** sends a kick message to the **Tea** task's **LAPSANG5** action, just to demonstrate the use of **DitsKick()**, as follows-

```
DitsKick(teaPath,"LAPSANG5",0,&transid,status);
```

The technique is very similar to sending an obey.

3.4.6 Exiting

The **CTest** exit action is slightly more complex then the **Tea** and **Coffee** tasks, as it also sends exit commands to these tasks, before causing itself to exit -

```
DitsObey(teaPath,"EXIT",0,0,status);
*status = STATUS__OK;
DitsObey(coffeePath,"EXIT",0,0,status);
*status = STATUS__OK;
DitsPutRequest(DITS_REQ_EXIT,status);
```

We don't supply a transaction id to the **DitsObey()** calls. As a result we cannot find out if the action fails, does not exist, etc. **DitsSend()** is useful in this case as we don't really want the complexity of waiting around for the subsidiary actions to exit, but be careful when using it.

If the path used in a call to **DitsSend()** (or any message sending routine) is either 0, or is invalid (the task died or the path was not found), then status will be set bad. By making use of this fact we can avoid working out if we have actually got a good path to send the **EXIT** action along.

3.5 User Interface Tasks

In the tasks we have seen so far, the **CTest** task starts actions in the **Tea** and **Coffee** tasks. Messages from the subsidiary tasks are sent to **CTest**. But how are the actions in **CTest** started and where at its messages sent?

There must be some task, some where in the system, which acts as a user interface. User interfaces to **Dits** must be able to-

- Accept commands from the user in an appropriate way (i.e. command line, windowing system etc.).
- After interpreting those commands, start actions in appropriate **Dits** tasks.
- Handle the responses returned by those actions, outputting messages for the user as necessary.

Dits provides the following features to support user interface construction-

- User interface context - Normally a task can only initiate messages to other tasks in the context of an action routine. This is necessary to ensure that messages from that action are directed to an appropriate place (the parent action). In a user interface there is no parent action. **Dits** provides special support for this by the provision of the **DitsUfaceCtxEnable()** routine. This routine enables a special context appropriate for user interfaces. The routine specifies a handler (or response) routine. When any transactions started after calling this routine complete the handler routine is invoked. The handler routine will receive all messages including error and informational messages. **DitsNotInterested()** cannot be used. For more details see [1]
- A default user interface response routine is provided. **DitsUfaceRespond()** can be specified as the argument to **DitsUfaceCtxEnable()** or it can be called directly as part of a user's response routine. This is normally done when the user considers the default handling (output of a message) sufficient for a particular message.
- The routines (**DitsMsgReceive()** and **DitsMsgAvail()**) can be called directly instead of via **DitsMainLoop()**. This allows appropriate interaction with other sources of input.
- The underlying message system may provide support of various message notification techniques.

The most basic user interface is one which sends a single action message to a task and waits for the completion message while outputting any error or informational messages sent.

A program to do this, **ditscmd**, is provided as part of **Dits**. To start the **BREW** action in **Ctest**, we would enter

```
ditscmd CTEST BREW
```

See [1] for more details on this command and its options [3].

[3] Note that under **VAX/VMS**, you should use `ditscmd "CTEST" "BREW"`. The quotes are needed to stop the VAXC run time library converting the task and action names to lower case. Under **VxWorks**, you should enclose all the arguments in double quotes, e.g. `ditscmd "CTEST BREW"` since arguments are not handled as neatly.

3.5.1 The Dui routines

Dits provides considerable flexibiltiy in user interface construction but it was found when building those user interfaces which are part of **Dits** that there is considerable in common between most user interfaces. This led to the **Dui** routines being developed.

As a result, this document does not go into the details of the **DitsUface** routines but instead looks at the the use of the **Dui** routines.

3.5.2 The Ditscmd program

We will have look at some of the details of the implementation of **ditscmd**. The full source is in appendix A.4.

ditscmd is complicated by having to handle command line options in two formats. Under **UNIX** or **VMS**, we use the **getopt()** routines. Under **VxWorks** we use the **DuiToken** routines[4]. As a result, the option handling differs depending on whether we need a **main()** routine or not.

The basic structure is as follows-

```
    DuiDetailsType details;
    .
    .
    .
    DuiDetailsInit(&details);
    details.MesasgeBytes = MESSAGEBYTES;
    details.ReplyBytes   = REPLYBYTES;
    details.MAXREPLIES   = MAXREPLIES;
    .
    .
    strncpy(details.TaskName,argv[optind++],sizeof(details.TaskName));
    strncpy(details.Action,argv[optind++],sizeof(details.Action));
/*
 *  If any arguments are supplied, get them and create an Arg structure.
 */
    if ((optind < argc)&&StatusOk(status))
    {
        register i;
        ArgNew(&details.ArgId,&status);
        for (i = 1; optind < argc ; optind++, i++)
        {
            char argname[20];
            sprintf(argname,"Argument%d",i);
            ArgPutString(details.ArgId,argname,argv[optind],&status);
        }
```

[4]Since the C run time library routine **strtok()** is not reentrant, we cannot use it.

```
}

details.ErrorHandler = ErrorHandler;
details.SuccessHandler = SuccessHandler;

DitsInit(thisTask,bufsize,0,&status);
DuiExecuteCmd(&details,&status);
DitsMainLoop(&status);

return(DitsStop(thisTask,&status));
```

The lack of a call to **DitsPutActionHandlers()** is because this program does not accept any obey messages from other tasks.

Most of the work here is by **DuiExecuteCmd()** using the items passed in the **details** structure. In this case, we specified the following-

MessageBytes	As per **DitsGetPath()** argument of the same name.
ReplyBytes	As per **DitsGetPath()** argument of the same name.
MaxReplies	As per **DitsGetPath()** argument of the same name.
TaskName	The name of the task to send the message to (including the node name, if any), in the format required by **DitsGetPath()**.
Action	The action or parameter name for the message
ArgId	An argument structure to be passed to the action
MsgType	The type of the message to be sent. This is one of **DITS_MSG_OBEY** Send an obey message. **DITS_MSG_KICK** Send a kick message. **DITS_MSG_GETPARAM** Send a get parameter message. **DITS_MSG_SETPARAM** Send a set parameter message. **DITS_MSG_CONTROL** Send a control message.

ErrorHandler

A routine to be called if the action completes with an error. The error handler looks like this-

```
int ErrorHandler(DuiDetailsType *details,
                          StatusType *status)
{
    DitsReasonType reason;
    StatusType reasonstat;

    if (*status != STATUS__OK) return(0);
    DitsGetReason(&reason,&reasonstat,status);
    DitsPutRequest(DITS_REQ_EXIT,status);
    if (*status == STATUS__OK)
        *status = reasonstat;
    return(1);
}
```

If this routine had returned 0, then a message would have been output using ErsOut, but nothing else. Since in **ditscmd** we don't expect any more messages, we want to request that the program exit. Additionally, by setting status to the value returned in DitsGetReason, DitsMainLoop will return with status set to this value. We can then pass this status to DitsStop where it is treated as the completion status of the program.

SuccessHandler

A routine to be called if the action completes successfully.
The success handler looks like this-

```
int SuccessHandler(DuiDetailsType *details,
                            StatusType *status)
{
    DitsArgType ArgId = DitsGetArgument();
    StatusType ignore = STATUS__OK;
    if (*status != STATUS__OK) return(0);

    if (ArgId)
    {
        char buffer[200];
        int length = 0;
        ArgToString(ArgId,sizeof(buffer),
                  &length,buffer,status);
        buffer[length] = '\0';
        if (*status == ARG__NOTSCALAR)
        {
            *status = STATUS__OK;
            MsgOut(status,
          "Non scaler argument returned");
        }
        else if (*status == STATUS__OK)
            MsgOut(status,buffer);
    }
    DitsPutRequest(DITS_REQ_EXIT,&ignore);
    return(1);
}
```

This is similar to the Error handler. The major difference we that we try to convert any argument value in the completion message to a string and output it.

The following items are also available in **details**

ArgFlag	What to do with the Argument. See the **flag** argument to **DitsPutArgument()**. Default value is `DITS_ARG_DELETE`
Node	Provides an alternative way for specifing the Name of the Node the target task is running on. If used, **TaskName** must not include a node specifcation.
TriggerHandler	Called when a trigger message is received.
InfoHandler	Called when an Informational message (generated by **Ers** or **MsgOut()**) is received.
CompletionHandler	Called whenever an action completes.
UserData	A pointer to void. The user can store anything required here and retrieve it in the various handler routines.
MaxMessages	As per the **DitsGetPath()** argument of the same name.
GetPathTimeout	The timeout to apply to get path operations. Default value of -1 = no timeout.
Timeout	The timeout to apply to actual messages. Default value of -1 = no timeout.

3.5.3 X-Windows interface

The **Dui** routines can also be used to build an X-Windows/Xt based user interface. The following code is taken from the **xditscmd** program, a simple X-Windows command interface for **Dits**.

```
extern int main(unsigned int argc, char **argv)
{
    StatusType status = STATUS__OK;         /* Routine status              */
    int condition;                      /* condition for XtAppAddInput  */
    int source;                         /* source for XtAppAddInput     */
    Widget      toplevel;               /* Toplevel widget              */

/*
 * Initialize the X toolkit, get top level widget.
 */
    XtSetArg(args[0], XtNallowShellResize, TRUE);
    toplevel = XtAppInitialize(&context,"Xditscmd",NULL,0,&argc,argv,
          fallback_resources,args,1);


    .
    .
    .
/*
 *  Setup Dits.
 *
 *  We initialise requesting X compatiblity and no sds for local messages.
 */
```

```
    DitsInit(thisTask,XdcBuffersGlobal(),DITS_M_X_COMPATIBLE|
                                          DITS_M_NO_LOCAL_SDS,&status);
    DitsGetXInfo(&source,&condition,&status);
/*
 *  Enable the routines for outputing Ers and MsgOut messages
 */
    DitsUfacePutErsOut(XdcErsOut.0,&status);
    DitsUfacePutMsgOut(XdcMsgOut,0,&status);
/*
 *  Handle dits initialise errors
 */
    if (status != STATUS__OK)
        return(DitsStop("XDITSCMD",&status));

/*
 *  Dits is Ready, set up the X windows/motif user interface
 */
    CreateMainWindow(toplevel,ConfigDialogEnabled);
/*
 * Realize top level widget - the user interface appears on the screen at
 * this point
 */
    XtRealizeWidget(toplevel);
/*
 *  Add Dits as another source of input with the routine InputFromDits
 *  to be called whenever a dits message is received.
 */
    XtAppAddInput(context,source,(XtPointer)condition,XdcInputFromDits,0);

/*
 *  Enter the main loop.  We should loop here forever - program exit is
 *  via the the Exit button which calls exit().
 */
    XtAppMainLoop(context);
/*
 *  Should never occur
 */
    return(DitsStop("XDITSCMD",&status));
}
```

We initialise **Dits** in the normal way, except that we specify the flag `DITS_M_X_COMPATIBLE` to **DitsInit()**. This flag tells **Dits** to use an X-Windows compatible message notification technique. We can then use **DitsGetXInfo()** to get the notification mechanism details. Later on, we use **XtAppAddInput()** to tell the X-Windows toolkit to add this mechanism to its list of inputs. Additionally, we specify to **XtAppAddInput()** a routine to be called when a **Dits** message is received - in this case - **XdcInputFromDits()**. This routine simply loops processing all the

available **Dits** messages before returning

```
extern void XdcInputFromDits (XtPointer client_data, int * source,
                           XtInputId *id)
{
    StatusType status = STATUS__OK;
    long int exitflag = 0;

    while (DitsMsgAvail(&status))
        DitsMsgReceive(&exitflag,&status);

    if ((status != STATUS__OK)||(exitflag))
        exit(DitsStop("XDITSCMD",&status));
}
```

This combination replaces **DitsMainLoop()**.

By default, messages output using **Ers** routines and **MsgOut()**, in the user interface, are output using **fprintf()**. In an X-Windows system, we would prefer they goes somewhere else. The routine **DitsUfacePutErsOut()** routine changes the output routine for **Ers** messages. The second argument to **DitsUfacePutErsOut()** is copied to the **outArg** argument when the output routine is called.

This is how **XdcErsOut()** is defined as-

```
extern void XdcErsOut (void * outArg, unsigned int count,
                        ErsMessageType messages[], StatusType * status)
{
    char text[ERS_C_LEN+1];
    register i;
    if ((*status != STATUS__OK)||(count <= 0)) return;
/*
 *  Setup the first message and output it.
 */
    strcpy(text,"##");
    strcat(text,messages[0].message);

    XdcMessageOutput(text,(messages[0].flags&ERS_M_HIGHLIGHT),
                       (messages[0].flags&ERS_M_BELL));
/*
 *  Output subsequent messages.
 */
    for (i = 1 ; i < count ; ++i)
    {
        strcpy(text,"# ");
        strcat(text,messages[i].message);
        XdcMessageOutput(text,(messages[i].flags&ERS_M_HIGHLIGHT),
```

```
                                (messages[i].flags&ERS_M_BELL));
    }
}
```

where XdcMessageOutput is a simple output routine which outputs into a message widget. This routine does not correctly handle the **ERS_M_ALARM** flag. This should probably be handled with an error popup as well as output to the scrolling area.

Likewise, the routine **DitsUfacePutMsgOut()** routine changes the output routine for **MsgOut()** messages and **XdcMsgOut()** is defined as-

```
extern void XdcMsgOut (char * string, void * client_data,
                                     StatusType  *status)
{
    if (*status != STATUS__OK) return;
    XdcMessageOutput(string,0,0);
}
```

So now we have handled incomming **Dits** messages and the output of messages to the user interface. The one thing left is actually sending commands. The source of commands would normally be some widget in the X-windows user interface. This example reads a command line and sends the message in this way-

```
/*
 *  Get space for the details and initialise it.
 */
    details = (DuiDetailsType *)malloc(sizeof(DuiDetailsType));
    DuiDetailsInit(details);
/*
 *  Get command and task name etc.
 */
    .
    .
    .
/*
 *  Setup message buffer size details.
 */
    details->MessageBytes = MessageSize;
    details->ReplyBytes   = ReplySize;
    details->MaxMessages  = MaxMessages;
    details->MaxReplies   = MaxReplies;
/*
 *  Get the message type and set the completion handler.  The completion
 *  handler is necessary to release the memory allocated for details.
 *  The SuccessHandler handles image structures only.
```

```
 */

    details->MsgType = MessageType;
    details->CompleteHandler = CompletionHandler;
    details->SuccessHandler = SuccessHandler;

    DuiExecuteCmd(details,&status);
```

Probably the most obvious changes are the use of **malloc()** to get space for the **details** variable and the setting of a completion handler, as well as a success handler. In addition, we don't set an error handler.

The completion handler, which will be called after the success handler, does nothing more then free the **details** structure -

```
static void CompletionHandler( DuiDetailsType *details , StatusType *status)
{
    free(details);
}
```

By doing this, we can have any number of command outstanding at one time.

Since we have added our own **MsgOut()** and **Ers** output handling routines and this type of program does not exit automatically when a message completes - we can use the default Success and Error handlers to output details of message completion for most messages. The exeception is for completion messages containing an Image Structure. This particular program can display Images and therefore handles them itself -

```
static int SuccessHandler( DuiDetailsType *details , StatusType *status)
{
    DitsArgType ArgId;
    if (*status != STATUS__OK) return(0);
/*
 *  Have we got an argument.
 */
    if (ArgId = DitsGetArgument())
    {
        char name[17];
        SdsCodeType code;
        long ndims;
        unsigned long dims[7];
/*
 *      If we, get the details.
 */
        SdsInfo(ArgId,name,&code,&ndims,dims,status);
        if ((*status == STATUS__OK)&&(strcmp(name,"ImageStructure") == 0))
```

```
        {
/*
 *          If we have an image structure, process it.
 */
            SdsFind(ArgId,"DATA",&ImageId,status);
            SdsInfo(ImageId,name,&code,&ndims,dims,status);
/*
 *          Valid image structure, display it.
 */
            if ((code != SDS_STRUCT)&&(ndims = 2)&&(*status == STATUS__OK))
            {
                MsgOut(status,
                "Action %s, Task %s, completed returning an Image Structure,",
                    details->Action,
                    details->TaskName);
/*
 *              Display the image.
 */
                .
                .
                .
                return(1);
            }
            else if (*status == STATUS__OK)
            {
                ErsOut(0,status,
        "Action %s, Task %s, completed returning an INVALID Image Structure,",
                    details->Action,
                    details->TaskName);
                return(1);
            }

        }
    }
    if (*status != STATUS__OK)
    {
        char errstring[100];
        MessGetMsg(*status,0,100,errstring);
        ErsOut(0,status,"Error handling completion of action %s - %s",
                    details->Action,errstring);
        *status = STATUS__OK;

    }
    return(0);
}
```

In the cases where we returned 1, we have handled this message ourselves. Where 0 is returned, we are requesting that the **Dui** routines handle the message completion.

If your user interface were more specific, you would probably have success handlers for each action initiated. For example, if you have a button which sends a particular command to a particular task, the success handler could look at the argument returned and display its value on the appropriate part of the user interface, returning 1 to say it has been handled.

The combination of **Dits** and X-Windows is quite a complex field and is still being developed. You are refered to [1] (Dui appendix) and [10].

3.5.4 Other Input Sources

In addition to combining **Dits** with X-Windows, it is possible to combine **Dits** with any other inputs of the type used with the X-ToolKit.

For example, you may want to use a file, such as the standard input device, as a source of input. If you are using the X-Windows toolkit, you can use **XtAppAddInput()** in the normal way.

When not using X-Windows, you should use the **DitsAltIn** routines. These work in a similar way but do not involve X-Windows. They allow you to add up to 10 alternative sources of input.

The technique is to-

- Initialise **Dits** with X compatibilty. (flag to **DitsInit()**).
- Initialise a variable of type **DitsAltInType** using **DitsAltInClear()**.
- Call **DitsAltInAdd()** for each alternative input source. You specify the variable above, details of your input source and a procedure to be invoked when input occurs on your source.
- Call **DitsAltInLoop()** instead of **DitsMainLoop()**.

See [1] for more details.

4 Parameter systems

A parameter system provides configuration and debugging support. These parameters do **NOT** correspond to command line arguments. They may be set and retrieved both externally and internally to the task.

Typically a parameter is set externally and retrieved internally to change the task configuration while those set internally and retrieved externally are used to indicate the state of task for debugging purposes.

External setting and retrieving of parameters is done from another task with **DitsSetParam()** and **DitsGetParam()**. Both these routines are very similar to the **DitsObey()** and **DitsKick()** routines. Instead of an action name, both routines take the name of a parameter in the target task.

In the case of **DitsGetParam()** the invoking action should reschedule to await a new entry with a reason of DITS_REA_COMPLETE or DITS_MESREJECTED. In the case of the former, the value of the parameter is in an **SDS** item id which can be accessed using **DitsGetArgument()**. It should have a similar to that created by the **ARG** routines. The item name for the **ArgGetx()** calls is the name of the parameter you requested.

In the case of **DitsSetParam()** you supply an argument which contains the new value for the parameter.

These messages are not seen by the application routines but are handled entirely by interaction between the Task's Fixed Part and the parameter system.

The parameter system is an optional part of **Dits** tasks. Additionally you can use any parameter system you wish, assuming it supports the appropriate interaction with the fixed part. The techniques for setting and getting parameter values internally to the task is dependent on the actual parameter system used.

4.1 The Simple Dits Parameter System

Dits provides a simple but useful parameter system, known as **Sdp**. **Sdp** uses **SDS** to store parameters in a format compatible to that used by the **ARG** routines.

The routine **SdpInit()** is used to initialise the parameter system, returning the **SDS** id of the parameter system. The routine **SdpCreate()** is used to create parameters while **Dits** is told about the parameter system by calling **DitsPutParSys()**. The following is an extract from the **main()** function of a **Dits** task -

```
static int one = 1;
static float two = 2.2;
static unsigned int three = 3;

static SdpParDefType params[] = {
        { "PARAM1", &one,       SDS_INT    },
        { "PARAM2", "hi there", ARG_STRING },
        { "PARAM3", &two,       SDS_FLOAT  },
        { "PARAM4", &three,     SDS_UINT   }};
int pardefsize = sizeof(params)/sizeof(SdpParDefType);


SdsIdType parsysid = 0;


SdpInit(&parsysid,&status);
SdpCreate(parsysid,pardefsize,params,&status);
DitsPutParSys(SdpGet,SdpSet,parsysid,&status);
```

The first item in each element of the parameter definition array (**params**) is the name of the parameter. The second item is an address of the initial value, while the third item is the type of the item. See [1] for more information.

You can define as many parameters as you want in the **params** array. Additionally, **SdpCreate()** can be called multiple times if necessary. **SdpGet()** and **SdpSet()** are the routines which will be called by the **Dits** fixed part when get and set messages are received.

Action routines can retrieve the value of **parsysid** using **DitsGetParid()** and then use **ARG** routines to access parameters.

See [1] and [3] for more details.

5 Object Orientated techniques

Earlier in this document it was mentioned that a task is seen as a software object to which messages are sent. By loading appropriate tasks you can build a complex system in an object oriented way.

Although the tasks themselves do not have to be written in an object oriented way, there is considerable gain to be made by doing so. This section examines how do do this.

The key to an object oriented approach in **Dits** is the **DitsPutActionHandlers()** routine. In section 3.2.1 we described how you set up the action definition array used by **DitsMain**. **DitsMain** supplies this array and its size in a call to **DitsPutActionHandlers()**. In a simple task you only call this routine once, but it can be called multiple times. When you do this any actions with the same name as actions already defined override the actions already defined. Any actions not redefined are left as is.

This allows us to create modules which implement certain sets of actions. For example one module may implement the actions which all tasks in a system should support - such as say **INITIALISE**, **POLL** and **EXIT**. Another module may implement actions for a class of tasks such as detectors, say **EXPOSE** and **READOUT**, while yet another module may implement actions specific to a particular detector. Modules may wish to override commands in earlier module, for example a particular detector system will normally want to implement its own **INITIALISE** action.

The **main()** routine of the actual task will draw all these modules together in the correct order by calling appropriate activation routines.

We shall look at an example which implements a simple spectrograph style task (**Sst**) which could be run from the AAO Observer system. The AAO Observer system requires that such a task support certain actions. The Generic instrument task package (**Git**, see [7]) provides default support for these actions. The complete source code for this example is in appendix A.5

5.1 The Simple Spectrograph Task

The main module of **Sst** is very simple-

```
#ifdef DITS_MAIN_NEEDED
    int main()
#else
```

```
    int SstMain()
#endif
{
   StatusType status = STATUS__OK;      /* Status variable          */

    SdsIdType parsysid = 0;             /* For parameter system id  */

/*
 *  First, initialse the parameter system and dits.
 */
    SdpInit(&parsysid,&status);
    DitsInit("SST",5000,0,&status);
    DitsPutParSys(SdpGet,SdpSet,parsysid,&status);
/*
 *  Activate the two module we use.  This enables action handlers and creates
 *  parameters
 */
    GitActivate(parsysid,&status);
    SstActivate(parsysid,&status);
/*
 *  Loop receiving messages.
 */
    DitsMainLoop(&status);
/*
 *  Shutdown dits and exit.
 */
    return(DitsStop("SST",&status));
}
```

The first thing we must do is initialise **Dits** and the parameter system. The call to **SdpInit()** initialises the parameter system, returning an identifier. This identifier is supplied to the call to **DitsPutParSys()**, along with the **Sdp** routines that **Dits** will use to respond to set and get messages.

We then activate each of the modules we are using. Each module should have a *pack***Activate()** routine. If the package adds parameters the parameter system identifier is passed to it, as occurs in both these cases. Since packages can override action and parameter names which already exist, we should call the activation routines in an order which will ensure we get what we want. Normally, for this type of task, **GitActivate()** is called first since it provides the default handlers for various actions. There is no limit to the number of packages you could bring together in this way.

When writing tasks in this fashion you should ensure your **main()** routine is in a separate file from your activation and associated routines. This will ensure your task can be included in another task at a later stage.

After the activation routines have been called we simply enter the main loop to receive messages.

5.2 Activation routines

The activation routine is often very simple, such as in **GitActivate()**-

```
extern void GitActivate (SdsIdType parsysid, StatusType *status)
{
    DitsPutActionHandlers(GitMapSize,GitMap,status);
    if (parsysid)
        SdpCreate(parsysid, GitParamCnt,GitParams,status);
}
```

All it needs to do is to add the actions and parameters appropriate to this module. By making the call to **SdpCreate()** conditional on a non-zero value for **parsysid**, we can use this module in tasks which do not wish to provide parameter system support. In this module **GdtMap** is defined as-

```
const DitsActionMapType GitMap[] = {
       {GitInit,      0,           0, "INITIALISE" },
       {GitNull,      0,           1, "CTRLC" },
       {GitNull,      0,           2, "DUMP_LOG" },
       {GitExit,      0,           0, "EXIT" },
       {GitNull,      0,           3, "LOG_LEVEL" },
       {GitPoll,      GitKPoll,    0, "POLL" },
       {GitNull,      0,           4, "POLL_PARAMETER" },
       {GitReset,     0,           0, "RESET" },
       {GitNull,      0,           5, "SIMULATE_LEVEL" },
       {GitNull,      0,           6, "UPDATE_NBD" },
       {DmonCommand,  DmonCommand, 0, "UMONITOR"}
       };
```

All the named actions are supported. Only the **POLL** and **UMONITOR** commands support kick messages. Most of the actions result in the routine **GitNull()** being called, which as is suggested, does nothing.

GitParams is defined as

```
static float Fone = 1.0;
static int   Izero = 0;

static SdpParDefType GitParams[] = {
        { "LOG_LEVEL",       "NONE",                 SDS_STRING },
        { "SIMULATE_LEVEL",  "NONE",                 ARG_STRING, },
        { "TIME_BASE",       &Fone,                  SDS_FLOAT },
        { "ENQ_DEV_TYPE",    "DITS_IDT",             SDS_STRING },
        { "ENQ_DEV_DESCR",   "Dits Generic Instrument task", ARG_STRING },
        { "ENQ_VER_NUM",     "P0.0",                 SDS_STRING },
        { "ENQ_VER_DATE",    "20-Nov-1992",          SDS_STRING },
        { "ENQ_DEV_NUMITEM", &Izero,                 SDS_INT  }};
```

The **Sst** module is set up in a similar way.

The **Git** package provides several other routines which are usefull to task implementers. See [7] for details.

6 Staging Library Routines

In a conventional package of library routines, each routine must be called entirely in the context of one stage of an action. In **DRAMA** it is possible to construct packages which implement operations in multiple action stages and then return control to the caller's code.

One of the best examples is getting a path to a task. In order to provide maximum flexibilty **DitsGetPath()** is complex to use, with a number of possible results which must be handled. What would work better in many cases is a package which gets the path and causes the next stage of your action to be invoked when the get path operation is complete or an error has occured. The **GitPathGetInit()** and **GitPathGetComp()** routines implement such a scheme. The first routine is called with details of the task to get a path. In addition you provide the address of an action handler routine which is to be invoked when the operation is complete. The second routine is called from the action handler to get the result of the operation. You will never see the interving action staging.

The useage of these routine is documented in [7]. We shall examine here the techniques which can be used to implement such routines.

The first thing to consider is where such a package stores information between action reschedules. section 7.2 examines some of the reasons we cannot just use static storeage. The best approach is probably to use **DitsPutActData()**. We must create a structure to contain the information and malloc space for it. In **GitPathGet** the structure looks like-

```
typedef struct {
    DitsPathType path;             /* Path to task                  */
    DitsActionRoutineType handler; /* User supplied handler         */
    void * client_data;            /* User supplied data            */
    void * old_actdata;            /* Value stored by DitsGetActData()*/
    StatusType status;             /* Status of operation           */
} Git___PathData;
```

Here, **old_actdata** is a indicator to the next problem we must address. The user may already be using **DitsPutActData()**. Before putting our own value with **DitsPutActData()**, we must get the current value with **DitsGetActData()** and save it in **old_actdata**. We will need to restore the original value as the last thing we do. The first path of **GitPathGetInit()** now looks like-

```
extern void GitPathGetInit(char * name, int MessageBytes, int MaxMessages,
                    int ReplyBytes, int MaxReplies, int timeout,
                    DitsActionRoutineType handler, void * client_data,
                    StatusType *status)
```

```
{
    void * old_actdata = DitsGetActData();  /* Get old action data  */
    DitsTransIdType transid;
    Git___PathData *Info;
    if (*status != STATUS__OK) return;

/*
 *  Get space for get path info.
 */
    Info = (Git___PathData *)malloc(sizeof(Git___PathData));
/*
 *  Store details.
 */
    Info->client_data = client_data;
    Info->old_actdata = old_actdata;
    Info->status = STATUS__OK;
    Info->handler = handler;
/*
 *  Dits will save this infomration for us.
 */
    DitsPutActData(Info,status);
}
```

Now we must initiate the getting of the path. The possibilties are

- Status is ok and we have the path immediately. Here we need to invoke the user's handler routine.
- Status is ok but we have to wait for the path. Here we need to setup the timeout, if required and wait for something to happen.
- Status is NOT ok. Again we need to invoke the user's handler routine.

The most consistent way to invoke the user's handler routine is to change the action handler using **DitsPutHandler()** and reschedule immediately. The overheard is not very large.

This is what the rest of **DitsPathGetInit()** looks like-

```
    DitsGetPath(name,MessageBytes, MaxMessages, ReplyBytes, MaxReplies,
                &Info->path, &transid, status);

    if (*status == STATUS__OK)
    {
/*
 *      If we already have the path, stage to the user's handler
 */
        if (transid == 0)
```

```
        {
            DitsPutHandler(handler,status);
            DitsPutRequest(DITS_REQ_STAGE,status);

        }
        else
        {
/*
 *          Wait for path.  Invoke stage 2 handler when this happens.  Setup
 *          timeout if it is required.
 */
            if (timeout > 0)
            {
                DitsDeltaTimeType dt;
                DitsDeltaTime(timeout,0,&dt);
                DitsPutDelay(&dt,status);
            }
            DitsPutHandler(Git___PathGetStage2,status);
            DitsPutRequest(DITS_REQ_MESSAGE,status);

        }
    }
/*
 *      Error, store the code and stage to the user's handler.
 */
    else
    {
        Info->status = *status;
        *status = STATUS__OK;
        DitsPutHandler(handler,status);
        DitsPutRequest(DITS_REQ_STAGE,status);
   }
```

If we have to wait for the path, then **Git__PathGetStage2()** is then next routine invoked. It must store details of the possible errors and then invoke the user's handler routine. This is what it looks like.

```
extern void Git___PathGetStage2(StatusType *status)
{
    DitsReasonType reason;
    StatusType reasonstat;
/*
 *  Get the Information structure and reason for entry.
 */
    Git___PathData *Info = DitsGetActData();
```

```
    DitsGetReason(&reason,&reasonstat,status);
    if (*status != STATUS__OK) return;
/*
 *  Set status based on reason for entry.
 */
    if (reason == DITS_REA_RESCHED)
        Info->status = GIT__PATH_TIMEOUT;

    else if (reason == DITS_REA_PATHFOUND)
        /* Nothing to do here */;

    else if (reason == DITS_REA_PATHFAILED)
        Info->status = reasonstat;

    else
        Info->status = GIT__PATH_INV_ENTRY;
/*
 *  Setup to call user's handler.
 */
    DitsPutHandler(Info->handler,status);
    DitsPutRequest(DITS_REQ_STAGE,status);
}
```

The next thing that happens is that the user's handler routine, specified to **GitPathGetInit()**, is called. The first thing this routine should do is call **GitPathGetComp()**. At this stage, the user must not assume the value of **DitsGetActData()** is what he put there (it is not) and must not put any value using **DitsPutActData()**. **GitPathGetComp()** is simple. All it needs to do is recover information for the user, restore **DitsGetActData()** and release the memory allocated for it's internal storage-

```
extern void GitPathGetComp(DitsPathType *path, void ** client_data,
                           StatusType *status)
{
    Git___PathData *Info = DitsGetActData();
    if (*status != STATUS__OK) return;
/*
 *  Restore the original action data.
 */
    DitsPutActData(Info->old_actdata,status);
/*
 *  Fetch the required values.
 */
    *path = Info->path;
    if (client_data)
        *client_data = Info->client_data;
```

```
    *status = Info->status;
    free((char *)(Info));
}
```

It should be possible to write very powerfull rescheduling packages using these tehniques.

7 Other facilities available

This section looks at some of the facilities available in **DRAMA** which have not been mentioned in the examples.

7.1 Action context

Action routines are called with a *context*. This can be one of the following-

Context	Meaning
DITS_CTX_OBEY	Normal action entry caused by either an Obey message or a reschedule for of an action.
DITS_CTX_KICK	The action entry caused by reception of a Kick message.
DITS_CTX_UFACE	User interface context. This only occurs after a call to **DitsUfaceCtxEnable()** or in a user interface response routine.

The current context can be fetched by calling **DitsGetContext()**. Context use useful when the same routine is used in handling Obey, Kick and/or User interface messages.

7.2 User and Action Data Routines

Under some operating systems, such as **VxWorks**, all tasks operate in the same memory address space. As a result global and local static variables are common to all tasks which call a module declaring a variable of a given name. To avoid problems with tasks clobbering each others variables **VxWorks** provides the **TaskVar** library. This library allows a variable to be saved and restored at each context switch. Although user written routines can use this library, its use makes your code **VxWorks** dependent and makes the task context switch time longer.

The normal technique is for one task variable to be added per task and for that variable to point to a dynamically allocated area of memory. All the variables which would otherwise be static/global will be placed in the dynamically allocated area. This is the technique used by **Dits** to protect its common data under **VxWorks**.

To avoid the user having to add another task variable, the **DitsPutUserData()** routine allows a user variable to be placed in the **Dits** common block. The user can then fetch this data back using **DitsGetUserData()**. Again the normal technique would be for the user to supply to **DitsPutUserData()** the address of a dynamically allocated structure.

Additonially, you can store and retrieve data specific to each action using **DitsPutActData()** and **DitsGetActData()**. Such data items are carried accross invocations of an action.

7.3 DitsInitiateMessage

We have seen the use of the routines **DitsObey()**, **DitsSend()** and **DitsKick()** to send messages to other tasks. These routines, together with others such as **DitsGetParam()** and **DitsSetParam()** are very similar. It would probably not surprise you to find out that these routines are all wrap ups of calls to a common sending routine. This routine is named **DitsInitiateMessage()**.

DitsInitiateMessage() has a more complex interface then the message sending routines built on it, but by using it directly, a bit more flexibility is available and more importantly, in situations where the same message is sent a lot of times, greater efficiency.

See [1] for the full description of **DitsInitiateMessage()**. What we are concerned with here is the **message** argument. **Message** is a structure of type **DitsGsokMessageType**. For the routines such as **DitsObey()**, which call this routine, this structure is constructed on the fly from the information in the call. In situations where the same message is sent a lot of times you should construct this structure once and then use **DitsInitiateMessage()** to send it. This could save considerable time in some situations.

7.4 Uface timers

To provide support for timeouts in user interface code **DitsUfaceTimer()** and **DitsUfaceTimerCancel()** are provided[5]. These routines may also be used by normal action code to implement multiple timeouts if required. The author cannot at the moment forsee the circumstances where they might be required, but if they are, the user's **ResponseRoutine()** should use **DitsSignal()** to signal an action.

8 Include Files

In general, each package has an include file defining data type and function prototypes for that package. You should include this file whenever you call routines from a package.

Dits is a bit more complex. Since a global **Dits** include file would be very large, **Dits** uses a number of include files. All modules using dits should include **DitsTypes.h** and **Dits_Err.h** to define the basic **Dits** types and error codes. The routine specifications in [1] lists for each routine the appropriate include file which must be included to use that routine. Since **Dits** makes considerable use of macros to increase speed it is very important to get the correct include file.

9 Compiling, Linking and Running

See the appropriate sections in [9] and [1] for details on how to compile and link and run **Dits** tasks.

[5]User interface code operating in UFACE context cannot use action reschedules to implement timeouts.

10 Compatibilty with Starlink-ADAM

Much AAO software is currently written using the Starlink - ADAM software environment. **DRAMA** provides several features which allow the mixing of Starlink and **DRAMA** in a system.

- The Adam to **Dits** interface task. [1] describes this task in more details. It allows ADAM tasks to communicate with **Dits** tasks. By using this task you can use ADAM user interfaces (such as ICL) to control **Dits** tasks and integrate **Dits** tasks into existing ADAM instrumentation systems.
- The **DRAMA** error reporting system (**Ers**) provides an alternative library which converts calls to **Ers** routines into calls to the Starlink **EMS** package. As a result packages written for **DRAMA** can be integrated directly into Starlink applications.
- When the appropriate way to do it becomes clear we will implement a technique which allows the Starlink **EMS** packages to output its messages via **Dits**. This will allows Starlink packages to be used in **DRAMA** tasks.

11 Software Organisation

This section to be rewritten when we have sufficient experience.

References

[1] Tony Farrell, AAO. *03-Aug-1993, Distributed Instrumentation Tasking System.* Anglo-Australian Observatory **DRAMA** Software Document 5.

[2] Keith Shortridge, AAO. *12-Oct-1992, Interprocess Message Passing System.* Anglo-Australian Observatory **DRAMA** Software Document 8.

[3] Jeremy Bailey , AAO. *9-Sep-1992, Self-defining Data System.* Anglo-Australian Observatory **DRAMA** Software Document 7.

[4] Tony Farrell, AAO. *19-Feb-1993,* **DRAMA** *Error reporting System.* Anglo-Australian Observatory **DRAMA** Software Document 4.

[5] Tony Farrell, AAO. *18-Feb-1993, A portable Message Code System.* Anglo-Australian Observatory **DRAMA** Software Document 6.

[6] Tony Farrell, AAO. *12-Feb-1992, UDISPLAY and the UMON routines.* Draft Anglo-Australian Observatory Software Document.

[7] Tony Farrell, AAO. *14-Apr-1993, Generic Instrumentation Task Specification.* Anglo-Australian Observatory **DRAMA** Software Document 9.

[8] Tony Farrell, AAO. *23-Dec-1992,* **DRAMA** *Software Organisation* Anglo-Australian Observatory **DRAMA** Software Document 2.

[9] Tony Farrell, AAO. *19-Jul-1993, Create Makefiles for* **DRAMA** *Programs* Anglo-Australian Observatory **DRAMA** Software Document 10.

[10] P.J.Asente & R. R.Swick. *1990, X Window System Tookkit* Digital Press X and Motif Series.

A Example code

This appendix gives the full source code listing for the various examples in the text.

A.1 Coffee.c

A.2 Tea.c

A.3 CTest.c

A.4 Ditscmd.c

A.5 Object Oriented programming examples

The Git Module

The Sst Module

The Sst_Main Module