ANGLO-AUSTRALIAN OBSERVATORY AAO/DITS_SPEC_5
DRAMA Software Report 5
Version 1.3

Tony Farrell
16-Nov-99

Distributed Instrumentation Tasking System

Contents

Revisions:

V0.6 03-Aug-1993 Change **DITS_REQ_ASTINT** to **DITS_REQ_SLEEP**. Introduce new routines (**DitsPutConnectHandler()**, **DitsPutDisConnectHandler()**, **DitsErrorText()**, **DitsPutKickHandler()**, **DitsPutDefObeyHandler()**, **DitsPutDefKickHandler()**, **DitsLosePath()**, **DitsUfaceTimer()** and **DitsUfaceTimerCancel()**.

Replace **DitsPutHandler()** by **DitsPutObeyHandler()**.

Add **client_data** arguments to **DitsUfacePutErsOut()** and **DitsUfacePutMsgOut()** routines.

Add **DitsLosePath()**.

Add timeouts to **DuiExecuteCmd()** structure.

Use **StatusType** for **stauts** arguments.

Note that you should now see [7] for details on building DRAMA programs.

V0.7 05-Jan-1994 Parameter Monitoring Supported. Task loading support incorproated. You can now change the handler for `DEFAULT` control messages. Changes to xditscmd to incorporate task loading and better X resources management. Routines to Set and Get parameters have been added to Sdp so that it will work with the parameter monitoring system.

The routine **DitsPutTaskId()** has been removed as it has fundamental problems. It is replaced by the pair **DitsEnableTask()** and **DitsRestoreTask()**.

The routines **DitsMonitor()**, **DitsMontiorMsg()**, **DitsMonitorDisconnect()**, **DitsMonitorTidy()** and **DitsPutParamMon()** have been introduced to support parameter monitoring.

The routines **DitsLoad()**, **DitsLoadErrorText()** and **DitsLoadErrorStat()** have been introduced to support parameter monitoring.

The enquires **DitsGetMsgLength()**, **DitsGetPathSize()** and **DitsGetTaskName()** have been added.

The routines **DitsDefault()** and **DitsPutDefaultHandler()** support the ability to change the handler for DEFAULT control messages.

Dits include files now support C++.

V0.8 22-Apr-1994 Orphaned transaction handling supported. Task deletion supported. Message Peeking Supported. Task Types and Descriptions supported. ADITS handles messages in both directions.

The routines **DitsDeleteTask()**, **DitsFindTaskByType()**, **DitsForget()**, **DitsGetTaskDescr()**, **DitsGetTaskType()**, **DitsPeek()**, **DitsPutOrphanHandler()**, **DitsSetDetails()**, **DitsTakeOrphans()** and **DitsTaskFromPath()** were added.

V0.9 23-Feb-1995 Document interrupt handler stuff.

Add new routines **DitsGetEntName()**, **DitsGetEntPath()**, **DitsGetEndReason()**, **DitsGetEntStatus()** and **DitsGetEndTransid()** which make access to the information previously returned by **DitsGetEntInfo()** easier to access.

Add **DitsGetTransData()** and **DitsPutTransData()** routines which allow a user specified data item to be associated with each transaction.

Add **DitsIsOrphan()** and **DitsGetParentPath()**.

V0.12 06-Dec-1995 Add `OFFSETS` item to **ImageStructure**. Added the new routine **DitsScanTasks()**. Added **DitsAppInit()** which allows the self path buffer size to be specified. Add **DitsRequestNotify()**.

Added support for blocking actions, with new routines **DitsActionWait()**, **DitsUfaceWait()** and **DitsPutEventWaitHandler()**.

Added support for multiple actions of the same name with routines **DitsPutActions()**, **DitsGetActIndex()** and **DitsSpawnKickArg()**. As part of a generalization needed to support this, the routines **DitsSignal()** and **DitsKill()** are withdrawn and replaced by **DitsSignalByName()** and **DitsKillByName()** (macros allow for backward compatibility). Added **DitsSignalByIndex()** and **DitsKillByIndex()**.

V0.13 19-Dec-1996 Add `DITS_M_NOEXHAND` flag to **DitsAppInit()** and **DitsInit()**. Incorporate Imp V1.0.

New routine **SdpPutStruct()** similar to **SdsPutSds()** but allows structures to be put into structures parameters.

Add routine **DitsTaskIsLocal()** which allows you to determine if a task is running on a local machine.

New routine SdpGetReadonly allows you to make a parameter system readonly from other tasks. New `SDP_` codes replace the use of `SDS_` and `ARG_` codes when setting up paramaters. **DitsPutParSys()** now done as part of **SdpInit()**.

A new flag, `DITS_M_REP_MON_LOSS` flag to **DitsInitiateMessage()** tells the monitor system to report monitors lost due to waiting for buffer empty notificatons.

DitsGetEntInfo()/rouDitsGetEntName now return the name of the subsidary actions when the reschedule reason involves such a transaction

Added **DitsPutCode()** to allow the code assocated with an action to be changed.

DitsPutActions() argument structure changed to provide an action cleanup routines.

DitsPathGet() replaces **DitsGetPath()**, proving new arguments The later is reimplemented in terms of the former.

DitsDisconnectRoutineType() now has a flags arguments. **DitsAppInit()** calling sequence has been changed to support new flags and a new structure to provide more details to initialisaction.

Many of the **DitsPut???Handler()** routines now return their original values. Where there was no client data item, then simply return the value and now application change is required. Where there was a client data item, new arguments have been added.

V0.15 04-Feb-1997 New routine **DitsAppParamSys()** replaces **DitsPutParSys()** and **DitsPutParamMon()**. MGET messages are not available (Multiple parameter gets). New routine **DitsArgNoDel()** allows Sds structures supplied by **DitsSignal()** calls to be left alone by DRAMA. Added **DitsPutPathData()** and **DitsGetPathData()** to allow data to be associated with paths.

V0.15.1 21-Apr-1997 Add new routine **DuiLogEntry()** which is used to log entries to an action handler using MsgOut. Add routine **SdpSetReadonly()**.

V1.2 23-Dec-1998 Add bulk data support routines, **DitsArgIsBulk()**, **DitsBulkArgInfo()**, **DitsBulkArgRelease()**, **DitsBulkArgReport()**, **DitsDefineSdsShared()**, **DitsDefineShared()**, **DitsDefineShared()**, **DitsGetEntBulkInfo()**, **DitsInitMessBulk()**, **DitsReleaseShared()** and **DitsTriggerBulk()**. Add section on bulk data. Version number of this document now matched DITS version number. Add new routines **DitsAltInDelete()**, **DitsTaskIsLocal()**, **DitsSpawnKickArgUpdate()**, **DitsSetParamSetup()**, **DitsSetParamTidy()**, **DitsUfaceErsRep()**,

V1.3 16-Nov-199] Add **DitsGetEntComplete()**.

1 Introduction

This document describes a tasking environment for use by real time instrumentation software running on a distributed system.

The tasking environment has many similarities with the version of ADAM used at the AAO. In particular, the techniques introduced by William Lupton and later modified by myself which make it a useful environment for writing complex control tasks. It does not attempt to reproduce ADAM although there is an interface between ADAM and the message system.

For an overview of the tasking model and examples of how to write tasks using this system, please have a look at [1].

This document is intended to provide detailed programmers documentation.

1.1 Dits

We assume here you are writing in the C programming language.

1.2 The Application's Main routine

The main routine of each task is responsible for calling:-

```
DitsAppInit(char *taskname(>),
         long int bytes(>),
         long int flags(>),
         DitsInitInfoType *initInto(>),
         StatusType *status)
DitsPutActionHandlers(long int size(>),
         DitsActionMapType *map(>),
         StatusType *status)
DitsMainLoop(StatusType *status)
DitsStop(char *taskname(>),
         StatusType *status)
```

DitsAppInit() is the initialization routine for the tasking system. It's taskname argument specifies the name this task should be known by to the system. The bytes argument specifies the size of the message buffer area. Flags determines various **Dits** options. "initInfo" allows various bits of informations, depending on the flags value, to be supplied. If not required, 0 can be specified.

DitsPutActionHandlers() specifies the default routines to be called when a message is received. An action's default routine is always called when an action starts (seq = 0) and for each reschedule unless overridden by a call to **DitsPutHandler()**.

DitsActionMapType is of the form:-

```
struct {
       DitsActionRoutineType obey;
       DitsActionRoutineType kick;
       long int code;
       char name[DITS_C_NAMELEN]; } map[]
```

That is, it is a structure array which associates an Obey and Kick routine with each action name. Additionally, a user specified code is specified. This code allows one routine to be called by various actions. By calling **DitsGetCode()**, the routine could determine which action is being called. (It could also call **DitsGetName()**, but this is less efficient).

When a message to initiate an action is received by the fixed part, the appropriate routine is called.

A call to **DitsPutActionHandlers()** is optional. If it is not made then the task does not support any actions. Normally only user interface tasks would omit it. It can be called multiple times. Later calls can redefine action names defined in earlier calls. This technique can be used to build new tasks out of the routines provided by other tasks in an object-oriented way. For example, a specific spectrograph task could be built using action handlers provided by a generic spectrograph task. An alternative interface **DitsPutActions()** is available. This allows more options to be specified.

The main routine should then call **DitsMainLoop()**. This routine is the main loop, which waits for messages and invokes the appropriate action routine when they come in.

1.3 Action concepts

The basic thread of control in a **DRAMA** task is the *Action*. A task receives *Obey* messages which cause a routine assocaited with a named *action* to be invoked. Such action routines will generally do some work as specified by the application author and then return.

An action routine may send messages to other tasks causing them to invoke action routines to do other jobs. Such tasks are known as subsidiary tasks and any action started is known as a subsidiary action. Actions may also do anything else that is required, such as controlling hardware devices or doing some processing.

After doing this, they may complete, causing a completion message to be sent to the originator or *reschedule* to await other message events. Such messages includes notifications of interrupts (using **DitsSignalByName()**), messages from the parent task (using **DitsKick()**) and messages from subsidiary actions (using **DitsTrigger()**).

When such a message is received, a routine (not necessarly, but possibly the original action routine) is invoked to handle the message. Again it can complete or reschedule after doing whatever work is necessary.

An action generally has a parent action, to which messages intended for the user interface are normally sent. The parent action has the job of forwarding these messages to the user.

By default, there may be only one action of a given name outstanding at any time. Additionally, by default, the action routines will reschedule by returning from the original routine in order to await future messages, after putting a rescheduling request. Both these restrictions may be broken at some additional cost in efficieny, see section 1.9 and section 1.10.

1.4 Arguments

Arguments may be passed to and returned from actions using a data item associated with each message.

Such arguments are always SDS items[1] , i.e., the initiator of an action creates an SDS item to contain the argument value and passes the SDS id of the item as the actual argument in appropriate calls. When an incoming message is received, the receiving action's actual argument is an SDS id, which can be used to obtain the argument values using SDS calls.

The ARG package, supplied as part of SDS and SDS itself, can be used to build, interpret and manipulate arguments, see [3].

1.5 Parameter system routines

The parameter system is provided by an entirely separate package so that it can be left out if desired. Also, a different parameter system could be used. These parameters do **NOT** correspond to command line arguments. They may be set and retrieved both externally and internally to the task.

The user should initialise the parameter system and pass an id which identifies it as the **parid** argument to **DitsAppParamSys()**.

The elements of the structure to be passed to this routine provide the interface between DRAMA and the parameter system. Routines such as those require to Get and Set parameter values are supplied.

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 with `SET` and `GET` messages sent from another task (alternatively, `MGET` messages allow you to get the values of multiple parameters in one message).

For the `GET` and `SET` message type, the associated message name is the name of a parameter of task while the value is an SDS item similar to that used by as command line arguments [2].

`GET` messages return the value of a task parameter while `SET` messages contain a value to which a parameter should be set.

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.

Techniques for internal getting and setting of parameter values depend upon the particular parameter system in use.

The parameter name `_ALL_` is defined to refer to the entire parameter system whilst `_NAMES_` is defined to return a list of all the parameter names. A `GET` message with these names specified

[1] SDS stands for "Self Defining Data System". It provides an efficient format for transferring data between machines with different data formats. SDS is described in [3]

[2] Although SDS is used to send parameter values about, it is not necessary for parameters to be stored by the parameter system as SDS items (although it might be easier)

as the parameter name will, if supported by the acutal parameter system retrieve the relevant item.

Althrough you can create your own parameter, system, most tasks will use the supplied parameter system, `SDP`. Instead of calling **DitsAppParamSys()** you invoked **SdpInit()** which calls **DitsAppParamSys()** itself. You then use **SdpCreate()** to create your parameters and various other `SDP` routines to get and set the values of the parameters.

If you don't wish to enable a parameter system, just don't call **DitsAppParamSys()**.

1.5.1 Parameter Monitoring

If supported by the parameter system and enabled by a task, parameters may be monitored. When the value of a parameter being monitor changes a message is sent to all tasks which requested that the parameter value be monitored.

Two types of parameter monitor are possible.

1. Regular monitoring will cause the originator of the monitor request to receive a `Trigger` message when the value of the parameter changes. The argument to this message will descibe the parameter name and it's new value. Regular monitor messages are started with a Monitor Type Message with the name `START`.
2. Forward monitoring causes an Obey message to be sent to a specified task when a parameter value changes.

Each monitor request to a particular task will have an id associated with it and you can use this to add or delete parameters to the list of those being monitored or cancel monitoring. See **DitsInitiateMessage()** for details.

1.6 Action context

An entry to an action may have one of two contexts. **DITS_CTX_OBEY** and **DITS_CTX_-KICKED**.

The **DITS_CTX_OBEY** context is the normal context. When an action is initiated by an `OBEY` message, its obey routine is called. **DitsGetContext()** will return a context of **DITS_-CTX_OBEY** and **DitsGetSeq()** a sequence of 0.

If the routine reschedules then the subsequent entry will again have a context of **DITS_CTX_-OBEY**, but the sequence is incremented by one for each reschedule.

An action awaiting reschedule may receive `KICK` message. Such a message will result in the kick routine being called. **DitsGetContext()** will return **DITS_CTX_KICKED**. Sequence will be incremented by one. There are three things it can do:-

1. Reject the message by returning a status code other then DITS__OK.
2. Cancel the action immediately. The routine should invoke the request **DITS_REQ_END**. (See section 1.7).

3. Reschedule the action. The action reschedules in the normal way.

An action may receive any number of `KICK` messages.

1.7 Requests

An action routine may request various things of the fixed part when the action returns to it.

The action routine calls **DitsPutRequest()** to register the request. If this routine is called more then once on an entry, then only the last call is significant. The form of this call is

```
DitsPutRequest(DitsReqType request, StatusType *status)
```

The following values are possible for request

DITS_REQ_END The action is to complete.

DITS_REQ_STAGE The action is to reschedule immediately.

DITS_REQ_WAIT The action is to reschedule on expiry of a timeout.

DITS_REQ_SLEEP The action is put to sleep. It may be rescheduled by being kicked or signalled.

DITS_REQ_MESSAGE The action is to reschedule on reception of a `RESCHEDULE` message.

DITS_REQ_EXIT The action is to complete and the task should then exit

Note, **DITS_REQ_WAIT**, **DITS_REQ_SLEEP**, **DITS_REQ_MESSAGE** are equivalent other then checks which are preformed. For example, with **DITS_REQ_WAIT**, an infinite timeout is not allowed.

If an entry to the action calls **DitsPutRequest()** with any request other then **DITS_REQ_-END** and status is ok then the final completion status of the action is ignored. That is, the action will not complete if it has a bad final status, if that action has already successfully requested some alternative course.

1.8 Task Types and Descriptions

A task may have a type and description. Task types are just integer values while descriptions are character strings. These items can be set using **DitsSetDetails()** and fetched for a named task using **DitsGetTaskType()** and **DitsGetTaskDescr()**.

You can determine the name of the first task of a given type. This scheme allow a system to setup classes of tasks. For example, you could define a class of Imager task. There may be several examples of such a task, with different features. The selection of a particular imager task may be up to the User, as may be the selection of the name of the task. By using the task type, you can work out the name of whatever imager the user has selected.

In addition, you can scan though tasks to find all tasks of a given type etc. See **DitsScanT-asks()**.

Types are not defined here, but after left to system designers.

1.9 Multiple actions of the one name

By default, only one invocation of an action of a given name may be outstanding at any one time. If you attempt to start a second action of the same name, you will get an error (`DITS__ACTACTIVE`). This style of programming is sensible for many cases of interest to **DRAMA** programmers. For example, you should only have one action attempting to move a single physical mechanism. Since the **ADAM** system on which **DRAMA** is based had this restriction, it is the default mode of operation of **DRAMA**.

There are through a class of problems where this restriction is unnecessary and others where it causes problems. Consider a general message logging task. Such a task may receive multiple messages of the same name but have to forward them onto a central logging machine. Hence it must have multiple actions of the same name outstanding at the same time. (There are ways around the probem, such as grabbing locks, but this adds complexity to the to both tasks).

To support multiple actions of the same name, use **DitsPutActions()** instead of **DitsPutActionHandlers()**. The call is the same, but the action map is of type **DitsActionDetailsType** instead of **DitsActionMapType**. This new type has the form

```
struct {
       DitsActionRoutineType obey;
       DitsActionRoutineType kick;
       long int code;
       long int flags;
       DitsSpawnCheckRoutineType spawnCheck;
       DVOIDP spawnData;
       char name[DITS_C_NAMELEN]; }
```

In addition to obey, kick, code and name elements, we have three new elements.

The flags argument is a set of flags to be assocaited with the action. If 0, then for the assocaited action, the entry is same as it would be with **DitsPutActionHandlers()**. The only other possible value at the moment is `DITS_M_SPAWNABLE`. If set, then this indicates the action should spawn, when an obey message is received, allowing mutiple action outstanding of the same name. Such spawned actions are otherwise identical to normal actions, except that it is harder to communication with them as will be explained below.

It may be neccessary for an application to limit the number of actions of the same name. If the `DITS_M_SPAWNABLE` flag is set, then `spawnCheck` is the address of a routine which is invoked before the action is invoked. If this routine returns with status ok, then the spawn is allowed. Otherwise, the bad status is returned to the parent action. If `spawnCheck` is not supplied, any number of actions of the specified name will be allowed - only limitied by system resources, in particular, memory.

The `spawnData` item is any user supplied item to be passed as the `clientData` argument to the `spawnCheck` routine.

As mentioned above, compilcations arise when you wish to communication with outstanding actions of the same name. The normal way of doing so under DRAMA is to send a **Kick**

message specifying the name of the action, or if within the same task, to *Signal* the action specifing the name.

If you may have multiple actions of the same name, the question arises about which one you are refering to. One way around this is by using the index of the action, which is unique to every action instance. This can be fetched using the **DitsGetActIndex()**.

The action index can be supplied to **DitsSignalByIndex()** to signal the action. Additionally, if sent somehow to another task, the index may be used as an argument to the kick of the action. This argument should be an integer item within a standard argument strucuture and be named "KickByIndex".

Alternatively, the parent action may use **DitsSpawnKickArg()** to create an argument which may be sent with the kick of the action. This requires the transaction id of the original obey message. (Whilst the parent action must create this SDS structure, the actual kick message including this argument may be sent from any task).

Spawnable actions are slightly less efficent to start up then non-spawnable actions. Some memory allocation and copying is required to extend the internal table of actions, although this is only done occasionally. In all cases, a search of a internal table is required and a copy of action details to the new entry. All this is additional overhead over non-spawnable actions.

1.10 Blocking actions to await messages

DRAMA programs support an event oriented style of programming. In this style, the normal approach is for the **main()** routine of an application to enter an event loop. The event loop then dispatches events to service routines, in this case action handler routines. Such routines then return to await future messages.

This type of programming is efficent in machine usage, but makes some applications much more complex. This complexity can be overcome if action routines could block to await messages while still allowing other messages to be processed. This can be done using the routine **DitsActionWait()**.

DitsActionWait() blocks the current action and waits for a message for it. Additional messages, which may invoke other actions, may be processed while waiting for a message to the original action. When a message is received for the original action, it is unblocked and control continues.

Using this routine adds some overhead to the rescheduling system and has some restrictions. These are as follows

1. If a subsequent action invokes **DitsActionWait()**, before the first has unblocked, then the subsequent action must be unblocked before the original one will be, i.e. "first in last out".
2. As a result of the above, there may be multiple messages waiting to be processed when the action is unblocked. It will have call the routine again to process them.
3. If messages received while the action is blocked have not been processed when the action complets, they are lost.

4. The action is NOT unblocked when a kick message is received. The kick handler will be invoked and may case the action to be rescheduled by other means.

5. The routine may be invoked from either Obey or Kick context, but not both at the same time.

If you don't use a standard **DRAMA** main loop routine, such as **DitsMainLoop()**, then you may need to call **DitsPutEventWaitHandler()** to specify a routine which will wait for **DRAMA** events.

1.11 Sending large amounts of data efficiently

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

When using the normal **DRAMA** message sending routines (**DitsTrigger()**, **DitsPutArgument()** and **DitsInitiateMessage()** (on which **DitsObey()** etc. are based)) **DRAMA** writes the message header directly into the receiving task's buffers. This is very efficient for messages without argument structures.

But, this operation requires that any argument structure you supply via a SDS id be exported from SDS into the buffer. This means that you must first put the data into an SDS item and then have **DRAMA** copy it into the buffer. For large amounts of Data, say image arrays, this is inefficient.

There are other problems. You are required to define the size of the largest message to be sent when you set up the path between two tasks. Also it is possible you need to read data from something like a frame buffer which can't be part of the DRAMA message buffers. All of these problems are solved by the "Bulk Data" feature.

This feature allows you to send very large amounts of data by specifying shared memory segments containing the data. For local transfers, this means no data is actually transferred, other then a small notification message sent to the target task. For network transfers, ony one area of memory is required on each machine. This approach has been used to efficiently transfer 150Mb between machines using DRAMA. The actual limit is determined by machine virtual memory restrictions. Bulk data techniques may place extra requirements on the receiving task, depending on the particular example.

1.11.1 Sending Bulk Data

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

Two **DRAMA** message operations can be replaced by "Send Bulk Data" equivalents - **DitsTrigger()** and **DitsInitiateMessage()**. (Note here that **DitsInitiateMessage()** is the underlying routine for **DitsObey()**, **DitsKick()**, **DitsGetParam()**, **DitsSetParam()**).

In each case, the basic procedure is the same. You first define an area of shared memory using **DitsDefineShared()** or **DitsDefineSdsShared()**. You then call the appropriate message sending routine specifying most of the normal arguments for the non-bulk routine and the shared memory segment details. When finished with the shared memory segement, you delete it using **DitsReleaseShared()**.

The bulk data equivalent of **DitsInitiateMessage()** is **DitsInitMessBulk()**. In addition to the normal arguments and flags of **DitsInitiateMessage()**, it has the extra shared memory details argument, a `NotifyBytes` argument and an extra flag - `DITS_M_SDS`. The new flag, if set, indicates that shared memory segment contains an SDS item which can be accessed by passing the address of the shared memory segment data area to **SdsAccess()**. If your shared memory segement does contain an SDS item, then setting this flag means that you can send you bulk data to any **DRAMA** task, which can handle is transparently as a normal SDS message argument. If you don't have SDS in your shared memory, the receiving task

The `NotifyBytes` argument, if non-zero, indicates that the initiating action should be notified every time that number of bytes is transferred. This is done by rescheduling the action with a reason of `DITS_REA_BULK_TRANSFERRED`. The transaction id associated with these reschedule events is that returned by **DitsInitMessBulk()**. You can fetch details of the progress of the target task in using the shared memory, allowing you to possibly start re-using that part of the shared memory. (although this depends on the design of the two tasks involved). Note that the reception of these messages is dependent on the behavious of the target task, which must invoked **DitsBulkArgReport()** at appropiate intervals. So don't rely on the reception of these messages.

In addition, you receive an entry with the code `DITS_REA_BULK_DONE` indicating the transfer has completed and the receiving task has finished with the data.

Reception of the `DITS_REA_BULK_DONE` message indicates you could reuse of free the shared memory segment if desired. You not not free the shared memory until you get the first `DITS_REA_BULK_TRANSFERRED` message or if you don't get any of these, the `DITS_REA_BULK_DONE` message.

When handling entries with a reason of or `DITS_REA_BULK_TRANSFERRED` or `DITS_REA_BULK_DONE`, the routine **DitsGetEntBulkInfo()** can be invoked to get details of the transfer, such as the progress.

The bulk data equivalent of **DitsTrigger()** is **DitsTriggerBulk()**. It takes five arguments, the shared memory details argument, an "sds" flag, a `NotifyBytes` item, a transaction id pointer and the normal status argument. The Shared memory details and `NotifyBytes` arguments are used as per **DitsInitMessBulk()**. The "sds" flag is just a logical item which is equivalent to the `DITS_M_SDS` flag to **DitsInitMessBulk()** - i.e. if true, the shared memory segment contains an exported SDS item.

The transaction id pointer is used to return the transaction id for the bulk data transfer. This is different from a normal **DitsTrigger()** operation where there is no transaction id as the operation completes immediately. You should expect to receive action reschedule events with a reason of `DITS_REA_BULK_TRANSFERRED` if you specified as non-zero value for `NotifyBytes`, until the bulk data is transferred. You should also expect an entry with a reason of `DITS_REA_BULK_DONE` when the target task has finished with the data.

When processing action entries with a reason of `DITS_REA_BULK_TRANSFERRED` or `DITS_REA_-BULK_DONE`, you can use **DitsGetEntBulkInfo()** to get status information of the transfer.

If the target task rejects the bulk transfer, you will get an action entry with reason `DITS_REA_-MESREJECTED`.

In the special case of sending bulk data arguments to a KICK message, the bulk data entries - `DITS_REA_BULK_TRANSFERRED` and `DITS_REA_BULK_DONE` - may be received after the message completion entry - `DITS_REA_COMPLETE`. This is because the target task may use **DitsBulkA-rgInfo()** in it's KICK handler but not call **DitsBulkArgRelease()** until a later invocation of it's OBEY handler. The sender cannot know the order involved as it is dependent on the design of the other task and if the other task is remote or local. The sender can use **DitsGetEnt-Complete()** to determine if the transaction is really complete.

Note that it is up to the sending and receiving tasks to agree on a protocol for access to shared memory. **DRAMA** just provides a facility for notification and network transfer.

Shared memory can be released using **DitsReleaseShared()**.

Note that by default **DRAMA** creates its bulk data in the same way as is done for the shared memory segements used by **DRAMA** for its own communications. The major implication of this is on UNIX machines where the bulk data normally created as mapped files in the IMP scratch directory. As a result, you must have sufficent disk space on that disk and you should note that if that directory is on a NFS mounted disk, there may be some delay in accessing the shared memory. See `http://www.aao.gov.au/drama/html/DramaTidbits.html#IMPfiles.` for more information.

Sending Example In this example, I map an area of shared memory and create an SDS object in it.

```
/*
 * First define the item used to store the shared memory objects.
 * This is required across reschedules so I have made it static.
 */

static DitsSharedMemInfoType SharedMemInfo;

void BulkTrigger(StatusType *status);
{
    SdsIdType id;
    unsigned long Size;
    void * Address;
    DitsTransIdType transid;
    if (*status != STATUS__OK) return;
/*
 *  Create the Sds template item and get the size it will have when
 *  defined.
 */
```

```
    CreateSdsItem(&id,status);
    SdsSizeDefined(id,&Size,status);

/*
 *  Create the shared memory using the size obtained above.  I am
 *  using a default format for the shared memory and
 *  asking DRAMA to create it for me.
 */
    DitsDefineShared(DITS_SHARE_CREATE, "", 0, Size,
                     1, &Address, &SharedMemInfo, Status);

/*
 *  Export the SDS item into the shared memory.  I am now finished
 *  with the original SDS item.
 */
    SdsExportDefine(id, size, Address, status);
    SdsDelete(id, status);
    SdsFreeId(id, status);

/*
 *  Access the exported SDS item and write the data to it.
 */
    SdsAccess(Address, &id, status);
    WriteSdsItem(id, status);
    SdsFreeId(id, status);
/*
 *  Send a bulk data trigger message, specifying the shared memory
 *  segment and notification every 1 Mb.
 */
    DitsTriggerBulk(&SharedMemInfo, 1, 1024*1024, &transid,
                    status);
/*
 *  Reschedule to await the bulk transfer messages.
 */
    DitsPutRequest(DITS_REQ_MESSAGE,status);
    DitsPutObeyHandler(HandleTransfer,status);
}

/*
 * This routine handles the bulk data transfer informational
 * messages.
 */
void HandleTransfer(StatusType *status);
{
/*
 *  Should only be triggered with a reason of DITS_REA_BULK_TRANSFERRED,
```

```
 *  DITS_REA_BULK_DONE or DITS_REA_MESREJECTED.
 */
    if (*status != STATUS__OK return;
    switch (DitsGetEntReason())
    {
        case DITS_REA_BULK_TRANSFERRED:
        {
            DitsBulkInfoType BulkInfo;
            DitsGetEntBulkInfo(&BulkInfo,status);
/*
 *          Not done, report progress
 */
            MsgOut(status, "Bulk transfer %.2f%% done",
                   (((double)BulkInfo.TransferredBytes /
                    (double)BulkInfo.TotalBytes) * 100.0));
            DitsPutRequest(DITS_REQ_MESSAGE,status);
            break;
        }
        case DITS_REA_BULK_DONE:
        {
            MsgOut(status, "Bulk transfer complete");
            DitsReleaseShared(&SharedMemInfo, 1, status);
            DitsPutRequest(DITS_REQ_END,status);
            break;
        }
        case DITS_REA_MESREJECTED:
        {
            DitsReleaseShared(&SharedMemInfo, 1, status);
            *status = DitsGetEntStatus();
            ErsRep(0,status,"Bulk data transfer rejected");
        }
        default:
            *status = DITS__APP_ERROR;
            ErsRep(0, status,
                   "Invalid entry to HandleTransfer function");
            break;
    }
}
```

Sending Example 2 In the above code, I showed how to create a shared memory segment which can be used for any data. In fact, as an SDS template was being written into the shared memory, you could use the following simpler call sequence.

```
/*
 * First define the item used to store the shared memory objects.
```

```
 * This is required across reschedules so I have made it static.
 */
static DitsSharedMemInfoType SharedMemInfo;

void BulkTrigger(StatusType *status);
{
    SdsIdType templateId,id;
    unsigned long Size;
    void * Address;
    DitsTransIdType transid;
    if (*status != STATUS__OK) return;
/*
 *   Create the Sds template item.
 */
    CreateSdsItem(&templateId,status);
/*
 *  Define a shared memory segement, exporting the template Sds
 *  structure into and returning the shared memory info and
 *  the SDS id used to access the exported SDS structure.
 */
    DitsDefineSdsShared(templateId, DITS_SHARE_CREATE, "", 0,
                 Size, 1,&id, &SharedMemInfo, Status);
/*
 *  I am finished with the template (but I could have used it again
 *  if I wanted to).
 */
    SdsDelete(templateId,status);
    SdsFreeId(templateId,status);

/*
 *  Write to the SDS item's data array.
 */
    WriteSdsItem(id,status);
/*
 *  In this example, I am now finished with the SDS id.
 */
    SdsFreeId(id,status);
/*
 *  Send a bulk data trigger message, specifying the shared memory
 *  segment and notification every 1 Mb.
 */
    DitsTriggerBulk(&SharedMemInfo, 1, 1024*1024, &transid,
                    status);

/*
 *  Reschedule to await the bulk transfer messages.
```

```
 */
    DitsPutRequest(DITS_REQ_MESSAGE, status);
    DitsPutObeyHandler(HandleTransfer, status);
}
```

1.11.2 Receiving Bulk data

DRAMA hides the details of bulk data from receiving tasks which don't wish to know, whilst at the same time allowing those tasks which wish to know to get proper access to the bulk data.

By default, if the bulk data item contains an SDS item, the receiving task need do nothing special to access it. A bulk data message looks like a normal Obey/Kick message or a response to a subsidy message started by the action. An action routine can use **DitsGetArgument()** to access the SDS item.

But if the bulk data does not contain an SDS item, then the underlying message is delivered but no argument will be seen (DitsGetArgument() will return 0). In this mode, if the sending task is local, it is sent the message which results in the `DITS_REA_BULK_DONE` entry when your action routine returns. If the sending task is remote, it gets this message when it's local transmitter task has finished forwardning the data.

1.11.3 Receiving Bulk data - special handling

Alternatively, the receiving task may want to handle the bulk data specially, depending on the application, say to allow it to be kept about or forwarded efficently to another task.

Here, what you may wish to do is to access non-SDS bulk data or to retain access to the bulk data after your action handler routine returns or to forward the bulk data to another task.

You can use the routine **DitsArgIsBulk()** to determine if your argument for this entry of your action routine is bulk data. If yes, then you can use **DitsBulkArgInfo()** to fetch details about the shared memory. Once you call this routine, you are in control of the shared memory and **DRAMA** will not try to delete it after your action handler routine returns. Additionally, **DitsArgIsBulk()** will now return false and any addition call to **DitsBulkArgInfo()** will return an error status.

When you are finished with the bulk data, call **DitsBulkArgRelease()** to release the bulk data. It is at this point that a local sending task task will receive the `DITS_REA_BULK_DONE` message. To indicate progress in processing the bulk data, you can call **DitsBulkArgReport()**. It is at this point that **DITS_REA_BULK_TRANSFERRED()** messages are sent to a local sending task.

An import point to note here is that whilst **DitsBulkArgInfo()** allows an action to be rescheduled but keep bulk data around, you must call **DitsBulkArgRelease()** at some stage **before** the action completes. A failure to do this may result in the sending task not getting correct notification of the bulk data transfer having completed - and therefore possibly failing to release the shared memory segement. If you attempt to complete the action before releasing the shared memory, **DRAMA** will automatically append an error report to your action completion and set status bad if it was not already bad.

Receive example In this example, I want access non-SDS bulk data.

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

    if (DitsArgIsBulk()
    {
        DitsBulkReportInfo    ReportInfo;
        void *Address;
        unsigned long Size;
        unsigned long NotifyRate;
        unsigned long usage;
/*
 *      Access the bulk data.
 */
        DitsBulkArgInfo(&ReportInfo, &Address,
                        &Size, &NotifyRate, status);
/*
 *      Use the bulk data, part 1.
 */
        UseBulkDataPart1(Address,Size,&usage,status);
/*
 *      Report on usage so far
 */
        DitsBukArgReport(usage, &ReportInfo, status);
/*
 *      Usage the bulk data part 2
 */
        UseBulkDataPart2(Address,Size,status);
/*
 *      Release the bulk data.
 */
        DitsBulkArgRelease(&ReportInfo, status);
    }
    else
    {
        /* Error , was expecting bulk data */
    }
}
```

1.12 Procedure Types

Various **Dits** procedures and data structures have parameters and fields that use procedure pointer types. For example, the action map takes action and kick procedures. These procedures return void (that is, return nothing) and take one argument of type pointer to long int. This

document defines procedures using ANSI C style typedefs. For example, the action procedure type required in the action map is defined as follows -

```
typedef void (*DitsActionRoutineType)(StatusType *status);
```

Thus the type of the procedure is `DitsActionRoutineType`, which is the same as a routine taking one argument and returning nothing. If you declare a variable that holds a pointer to an Action Routine, you declare it as-

```
DitsActionRoutineType ActRoutine;
```

However, the procedure itself has the return type `void`:

```
void ActRoutine(StatusType *status)
{
...
}
```

Declaring the action procedure as an DitsActionRoutineType would incorrectly mean that the procedure returns a pointer to an action routine. See appendix A for a list of all procedure types.

1.13 Interrupt Handlers

Support is provided for communication between interrupt handlers and main line code. Normally, interrupt handlers cannot use Dits routines as they do not have the required task context. What you normally want to do in interrupt handlers is cause a reschedule of an action in the main line code. This can be done using **DitsSignal()**. But, in order for **DitsSignal()** to work, you must setup the task context. First your code must grab required information in its main line code-

```
DitsTaskIdType TaskId = DitsGetTaskId();
```

This can be done any type after **DitsAppInit()** has been invoked. You must then pass this TaskId to you interrupt handler, using whatever technique is provided by your interrupt system.

Now before calling any Dits routines, your interrupt handler should enable Dits, suppling the task Id obtained above to **DitsEnableTask()**. After calling the approiate routine (normally **DitsSignal()**) it should restore the original context of the interrupt handler using **DitsRestoreTask()**)-

```
DitsTaskIdType SavedId;
StatusType status = STATUS__OK;
DitsEnableTask(TaskId,&SavedId);
DitsSignal("MY_ACTION",0,&status);
DitsRestoreTask(SavedId);
```

Note that none of this is necessary on systems where the "Interrupt" takes place in the context of the task (such as VMS AST routines and UNIX Signals). In these cases, you can just invoke **DitsSignal()** directly.

See the appropriate routine descriptions for more details.

2 The DITS routines

The `DITS` (Distributed Instrumentation Tasking System) routines allow a task to control its own behaviour and interact with other tasks.

2.1 DITS system routines

These routines setup and invoke `DITS`. You should include the file `DitsSys.h` to use these routines.

Routine	Function
`DitsAppInit`	Initialise Dits (as per **DitsInit()** but you can specify the self path buffer size.
`DitsGetTaskId`	Restore task id.
`DitsEnableTask`	Enable task in interrupt handler.
`DitsInit`	Initialise Dits.
`DitsMainLoop`	Dits main loop.
`DitsPutActionHandlers`	Register action handlers.
`DitsPutActions`	Register action handlers (has more options).
`DitsRestoreTask`	Restore interrupted task status in interrupt handler.
`DitsSetDetails`	Set the tasks type and description.
`DitsStop`	Shutdown dits.

2.1.1 Lower level access to `DITS`

While sufficient for most situations, DitsMainLoop is not always appropriate. The following routines provide the user with more control, allowing for example, multiple sources of input. You should include the file `DitsSys.h` to use these routines.

Routine	Function
`DitsGetXInfo`	Returns information needed by X-windows to coexist with **Dits**.
`DitsMsgAvail`	Returns the count of available messages.
`DitsMsgReceive`	Receive and process the next message, blocking if necessary.
`DitsPutEventWaitHandler`	Supply a routine to be used to test for **DRAMA** messages when an action blocks using **DitsActionWait()** or **DitsUfaceWait()**.

In addition, the following routines can be to build **Dits** tasks with sources of inputs other then **Dits** messages.

Routine	Function
`DitsAltInAdd`	Add an alternative input source.
`DitsAltInDelete`	Delete an alternative input source.
`DitsAltInClear`	Clear a variable of type **DitsAltInType**.
`DitsAltInLoop`	Implements a **Dits** main loop with alternative sources of message events.

See `DitsAltInLoop` for more details.

2.2 Interaction with fixed part

These routines get details of, or modify, the behaviour of the action which invokes them. You should include the file `DitsFix.h` to use these routines.

Routine	Function
`DitsArgNoDel`	Tell the fixed part not to delete the argument to this action entry
`DitsDeltaTime`	Create a delta time usable by **DitsPutDelay()**
`DitsGetActData`	Fetch the item stored with **DitsPutActData()** for this action.
`DitsGetActIndex`	Get the index of the current action.
`DitsGetArgument`	Get the argument supplied to the current invocation of the current action.
`DitsGetCode`	Get the code associated with the current action (see **DitsPutActionHandlers()**).
`DitsGetContext`	Get the context of the current action.
`DitsGetName`	Get the current action's name.
`DitsGetReason`	Get the reason for the entry of the current action.
`DitsGetSeq`	Get the current action's sequence number.
`DitsGetTaskName`	Return the name of the current task.
`DitsGetUserData`	Fetch the item stored with **DitsPutUserData()**.
`DitsPrintReason`	Output details of the reason for the current entry (to assist in debugging).
`DitsPutActData`	Store an item in the `Dits` data block associated with the current action.
`DitsPutArgument`	Put the argument to be returned by the current action on completion.
`DitsPutCode`	Change the item associatd with an action an retrieved by **DitsGetCode()**.
`DitsPutDefKickHandler`	Put the default kick routine to be called when an action is initiated.
`DitsPutDefaultHandler`	Put the routine which is invoked by the fixed part in response to a control message with name `DEFAULT` (change default directory).
`DitsPutDefObeyHandler`	Put the default obey routine to be called when an action is initiated.
`DitsPutDelay`	Put the delay/timeout before rescheduling the current action.
`DitsPutKickHandler`	Put the routine to be called for a kick message directed at the current action.
`DitsPutObeyHandler`	Put the routine to be called for the next reschedule of the current action.
`DitsPutRequest`	Indicates what the fixed part should do when this action returns. See section 1.7.
`DitsPutUserData`	Store an item in the `Dits` data block.

2.3 Interaction between actions in the one task

These routines allow events within a task to trigger the rescheduling of another action within the current task or kill another action. You should include the file `DitsSignal.h` when using

these routines.

Routine	Function
`DitsKillByIndex`	Kill an action immediately without further rescheduling using the action index (see **DitsGetActIndex()**).
`DitsKillByName`	Kill an action immediately without further rescheduling using the action name.
`DitsSignalByIndex`	Trigger the rescheduling of an action using the action index (see **DitsGetActIndex()**).
`DitsSignalByName`	Trigger the rescheduling of an action using the action name.

To actually start or kick actions in the same task, the routines in section 2.4 can be used.

2.4 Interaction with other tasks

All tasks work by receiving messages from other tasks. When an `OBEY` or `KICK` message is received, the details of the initiator are remembered and any responses (action completion, error messages etc) are sent to the initiator.

The job of some tasks will be to control other tasks. Messages sent to other tasks are of two types. The first type are expected to generate responses. Routines which initiated such messages will create and return a transaction id (transid) which can be used to identify the transaction. Other messages will generate no response.

When an action of a task initiates a subsidiary action (which may or may not be in another task), then the originating task's fixed part can expect messages from the subsidiary action which will normally cause the action to be rescheduled. Various routines will provide information about why the action has been rescheduled, both in this section and the previous section.

Note that a task may send messages to itself. A initiator action should not complete before its subsidiary action, therefore there is somewhere for the error messages to be sent.

All message system interaction is asynchronous (does not block) and the task must return to the fixed part to await messages.

Note that you can associate a data item with each transaction using the **DitsPutTransData()** routine and retrive it later with **DitsGetTransData()**. Similar routines exist for paths, **DitsPutPathData()** and **DitsGetPathData()**.

You can request notification of a target tasks buffer being empty by using **DitsRequestNotify()**.

2.4.1 Orphaned Transactions

In the scheme described above, every action has a parent action. This ensures that messages from subsidiary have some where to go (it's parent).

There are two problems with this scheme. The first concerns user interfaces. Thse are is not of concern to most programmers. It is addressed by the routines in section 2.7 and described in [1].

The second problem concerns what happens if an action, either deliberately or due to a programming error, exits before all of its subsidiary actions have completed. In this case, the subsidiary actions are called **Orphaned Transactions** (orphans).

When an orphan completes (or sends trigger or informational messages), the task which contained the parent action is sent a message. Since the action no longer exists, it cannot be sensibly delivered. There are three possibilites for handling such a case

1. By default, a simple message is written to stderr describing the message. This technique is not very nice but if you don't expect orphans to occur and stderr will always be accessible, it is probably good enough.
2. You can use **DutsPutOrphanHandler()** to specify a routine which is invoked when an orphan transaction completes. The routine is called in UFACE context (see section 2.7 and [1] for details of UFACE context). It allows a very general response to the problem.
3. You can reparent orphans. The routine **DitsTakeOrhans()** allows an action to take over orphaned transactions, such that they are considered subsidiary actions of the new action.

2.4.2 Peeking

Sometimes, it is not possible to program a task such that it returns to the fixed part regularly. This style of programming does not work well with event driven systems such as DRAMA. It tends to make the system unresponsive and makes it hard to Cancel operations.

DRAMA provides two routines which help in such suitations. The first routine, described eariler - **DitsMsgAvail()** - returns the count of outstanding messages. This allows an action to work until it sees that a message has come in, and then reschedule.

The above mode of operation is suitable for some jobs but not all. In some jobs, it is not practical to reschedule at all, but you will want to accept abort commands. The routine **DitsPeek()** allows you to peek at incomming messages. It can pick out messages of interest to you and return their details. For example, you can make it pick out KICK messages to your action and fetch the arguments of the KICK. From that you can work out if you should abort. **DitsPeek()** is quite complex. When called, it will handle any message it can (such as parameter get and monitor messages). This makes the task look quite responsive inspite of the lack of rescheduling.

2.4.3 The routines

The following routines are used for intertask communication, you should include the file `DitsInteraction.h` when using them (For **DitsPeek()**, use `DitsPeek.h`). Also see the section on bulk data.

Routine	Function
`DitsActionWait`	Block action to wait for a message.
`DitsCheckTransactions`	Check outstanding transactions for the current action.
`DitsFindTaskByType`	Find a task given its type.
`DitsGetEntInfo`	Get information about this entry.
`DitsGetEntName`	Get name associated with this entry.
`DitsGetEntPath`	Get path associated with this entry.
`DitsGetEntReason`	Get reason associated with this entry.
`DitsGetEntStatus`	Get status associated with this entry.
`DitsGetEntTransId`	Get transaction id associated with this entry.
`DitsGetMsgLength`	Return the length of a message.
`DitsGetParentPath`	Returns the path to the task which is initiated this action.
`DitsGetPath`	Returns a path to a specified task. Also see **DitsPathGet()**
`DitsGetPathData`	Get data accoaited with a path.
`DitsGetPathSize`	Return the size of the buffer to the task the path to which is specified.
`DitsGetTaskDescr`	Return a task's description.
`DitsGetTaskType`	Return a task's type.
`DitsGetTransData`	Get data associated with a transaction.
`DitsInitiateMessage`	Initiate a message system transaction with another task.
`DitsInterested`	Indicate interest in messages.
`DitsKick`	Send a `KICK` message to a task.
`DitsLosePath`	Mark a path as lost to force **DitsPath()** to try again at the underlying message system level.
`DitsNotInterested`	Indicate lack of interest in messages.
`DitsObey`	Send an `OBEY` message to a task.
`DitsPutPathData`	Assocaite a data item with a path.
`DitsPathGet`	Get a path ot a task, a more complete interface then **DitsPathGet()**.
`DitsPeek`	Peek at incomming messages.
`DitsPutConnectHandler`	Put a routine to be called when another task attempts to connect to this task.
`DitsPutDisConnectHandler`	Put a routine to be called when another task disconnects from this task.
`DitsPutTransData`	Assocaite a data item with a transaction.
`DitsRequestNotify`	Request notification of a task's buffer being empty.
`DitsScanTasks`	Scan for tasks.
`DitsSpawnKickArg`	Create an argument to be used for kicking spawned actions.
`DitsSpawnKickArgUpdate`	Update an argument created by the above routine to use a new transaction id.
`DitsTaskFromPath`	Return the name of a task given a path to it.
`DitsTrigger`	Trigger the parent of the current action.
`DitsTaskIsLocal`	Indicate if a task is local or remote.

2.5 Task Loading

Routines used in loading a task via **Dits**. See **DitsGenEntInfo()** for details of responses to **DitsLoad()**. It is also possible to delete a known task.

Routine	Function
`DitsDeleteTask`	Delete a known task.
`DitsLoad`	Load a specified task.
`DitsLoadErrorStat`	Return load system specific error status.
`DitsLoadErrorText`	Return a text version of the system specific error status.

2.6 Orphan Handling

These routines enable you to handle orphaned transactions in a controlled way. Include `DitsOrphan.h`.

Routine	Function
`DitsForget`	Explictly make transaction an orphan.
`DitsIsOrphan`	Indicates if a transaction has been orphaned.
`DitsPutOrphanHandler`	Set the routine which is invoked when an orphaned transaction completes.
`DitsTakeOrphans`	Reparent Orphans.

2.7 User interface construction

Construction of user interfaces is supported by the following routines.

Routine	Function
`DitsUfaceCtxEnable`	Enable user interface context for the current task.
`DitsUfaceErsRep`	Given a SDS id for a ERS message structure returned by a subsidary task, report it using **ErsRep()**.
`DitsUfacePutErsOut`	Put the routine to be called for **Ers** message output in user interface context.
`DitsUfacePutMsgOut`	Put the routine to be called for output resulting from calls to **MsgOut** in user interface context.
`DitsUfaceRespond`	General Uface context response routine.
`DitsUfaceTimer`	Setup a timer.
`DitsUfaceTimerCancel`	Cancel a timer.
`DitsUfaceWait`	Block the current thread of control and await a message.
`DitsUfaceWaitComp`	Tidy up after a call to **DitsUfaceWait()**.
`DitsUfaceWaitInit`	Setup for a call to **DitsUfaceWait()**.

See section 9 for more details.

2.8 Bulk data routines

This routines are involved in sending or receiving bulk data or creating the shared memory used for bulk data transfer.

Routine	Function
`DitsArgIsBulk`	Indicates if the argument to and action is bulk data.
`DitsBulkArgInfo`	Returns details about a bulk data shared memory argument.
`DitsBulkArgRelease`	Notify **DRAMA** you are finished with a bulk data shared memory argument.
`DitsBulkArgReport`	Notify **DRAMA** about progress in the use of a bulk data shared memory argument.
`DitsDefineSdsShared`	Define a shared memory segment containing and SDS structure and export from a template into the segment.
`DitsDefineShared`	Define a shared memory segement.
`DitsGetEntBulkInfo`	Retrieve details about a bulk data transfer.
`DitsInitMessBulk`	Send a bulk data message.
`DitsReleaseShared`	Release a shared memory segement.
`DitsTriggerBulk`	Send a bulk data trigger message.

2.9 Parameter system interaction

These routine all access to the parameter system of the invoking and other tasks. You should include the file `DitsParam.h` when using them.

Routine	Function
`DitsGetParam`	Send a message to get the value of the specified parameter in a task.
`DitsGetParId`	Get the value supplied as the **DitsPutParSys() parid** argument.
`DitsAppParamSys`	Enable/change a parameter system.
`DitsSetParam`	Set the value of a parameter in another task.
`DitsSetParamSetup`	Used to set up a `DitsGsokMessageType` structure for a parameter set message with a long parameter name.
`DitsSetParamTidy`	Tidy up after the previous call.

2.10 Parameter Monitoring

These routines support parameter monitoring. This routines are normally only of interest to writers of parameters systems, which is they only spot they are normally called. See the routine **DitsInitiateMessage()** for details of how user's use parameter monitoring.

Note, parameter monitoring is enabled useing **DitsAppParamSys()**. The routines here are used to implement it.

Routine	Function
`DitsMonitor`	Called by a parameter system to indicate a parameter value has changed.
`DitsMonitorDisconnect`	Invoked by Dits to indicate a task has disconnected and any monitoring involving that task should be cancelled.
`DitsMonitorMsg`	Invoked by Dits to indicate a message of type monitor has been received.
`DitsMonitorTidy`	Invoked by Dits when the task is shutting down (DitsStop invoked) to allow the monitor system to tidy up.

2.11 Utility Routines

This section contains various utility routines provided as a convenience to users.

Routine	Function
`DitsDefault`	Change the default directory. This routine is the default routine invoke when a control message of type `DEFAULT` is invoked. **DitsPutDefaultHandler()** can change this, but the user may want to call this routine to do the actual work.
`DitsErrorText`	Returns a pointer to a text string describing and error code.
`DitsNumber`	Return the size of an array.
`DitsGetSymbol`	Returns the value of an environment symbol, such as an environment variable, VMS logical name or Window registry entry.
`DitsSetDebug`	Control the internal debugging flags.

2.12 Obsolete Routines

The routines here have been made obsolete by the specified alternative. There is no intention of removing any of these routines but they are often an efficent implementation since they will be implemented in terms of the new routine.

Routine	Function
`DitsInit`	Use **DitsAppInit()**, which is a more complete interface.
`DitsGetPath`	Use **DitsPathGet()**, which is a more complete interface.
`DitsPutActionHandlers`	The new routine is **DitsPutActions()** which is more complete but the former is not really considered obsolete, just degreaded as it is a better chose for simpler cases.
`DitsPutParSys`	Use **DitsAppParamSys()** which is a more complete interface. Also note that **SdpInit()** now calls this routine itself so the call of this routine can normally be removed from your code.
`DitsPutParamMon`	Use **DitsAppParamSys()** which is a more complete interface.
`DitsSignal`	Use **DitsSignalByName()** which is an identical implementation. The new name distingishes it from **DitsSignalByIndex()**.
`DitsKill`	Use **DitsKillByName()** which is an identical implementation. The new name distingishes it from **DitsKillByIndex()**.

3 Sending messages to the user interface

The **MsgOut()** routine can be used to send messages to the user interface. You should include the file `DitsMsgOut.h`.

In the furture, **MsgOut()** will be replaced by the **Mrs** package which will provide more of the facilities provided by **Ers** (below). When this happens, **MsgOut()** will continue to work.

4 Error reporting

`Dits` makes use of the Drama error reporting system (**Ers**). See [4] for more details. Any unflushed messages will be flushed when an action returns.

5 Sdp - Simple dits parameter system

The `Sdp` routines, implement a simple parameter system for use by Dits tasks. The following routines are provided.

Routine	Function
`SdpInit`	Initialise a parameter system and return the identifier to it. This identifier should be supplied to `DitsInit` as the `parid` argument.
`SdpCreate`	Create parameters. Any array of parameter definitions is supplied. This routine can be called multiple times.
`SdpGet`	Returns, as an SDS id, a parameter value. This routine is intended to be supplied to `DitsInit` as the `GetRoutine` argument.
`SdpSet`	Sets a parameter value, given an SDS item containing the new value. This routine is intended to be supplied to `DitsInit` as the `SetRoutine` argument.
`SdpSet`x	Set a parameter value from inside a task. x may be one of the following, indicating the type of the value supplied.
`SdpSetReadonly`	Sets the parameter system as read only by other tasks.
	c A character item.
	d A double size floating point value.
	f A float size floating point value.
	i A long integer.
	i64 A 64 bit integer.
	s A short integer.
	u An unsigned integer.
	us An unsigned short integer.
	u64 A 64 bit unsigned integer.
	String A character string.
	Sds An Sds item.
	Conversions are preformed, is possible to the type of the actual parameter.
`SdpGet`x	Get a parameter value from inside a task. x is as per **SdpSet**x**()** above, indicating the type we want the value in. When the type is `Sds` then the Sds id of the actual parameter is returned, allowing the parameter to be updated directly with Sds calls.
`SdpUpdate`	When **SdsGetSds()** has been used to obtain access to the sds id of a parameter, this routine can be used to notify the parameter system that the parameter value has been changed. This call is necessary to ensure parameter monitoring works as expected by other tasks.

These routines provide you with the ability to create the parameter system and the necessary arguments to `DitsInit`. They do not provide you with the ability to manipulate the parameter system from your own routines.

This can be done using the `ARG` package provided with SDS, or with SDS itself (see [3]. Use the value returned by `SdpInit` as if it had been created by `ArgNew`[3].

6 Running Dits tasks

It is possible to run Dits tasks on a single machine by simply running the program. No setup is required. When networking is required, then the network tasks required by underlying message system - Imp [2] - must be run. This is done as follows -

Under VMS Execute the command `DRAMASTART` [4]

Under UNIX Execute the following commands [5]

```
~drama/dramastart
source $DRAMA/drama.csh
```

If you have bourne style shell, replace the second line by "`. $DRAMA/drama.sh`".

Under VxWorks This depends on how the VxWorks system has been set up. See [6] for more details.

In all systems you can now invoke `dits_netstart` to startup the network. Use `dits_netclose` to shut it down.

7 ADAM to Dits interface

It is possible for VMS ADAAM tasks to initiate actions in Dits tasks. ADAM tasks can of course call Dits routines directly, but to allow existing ADAM tasks to communicate with Dits tasks, a message conversion program - ADITS - has been provided.

From the point of view of ADAM tasks, the ADITS program is an ADAM network task. When ADITS receives ADAM messages, it translates them to appropriate Dits messages and forwards them to the appropriate Dits task. Replies from Dits tasks are sent to ADITS which converts them to ADAM messages and send them to the originator of the original message.

From the point of view of Dits, ADITS is an IMP Translator task. If the IMP message system cannot locate a task, it will ask any translator tasks if they handle the task.

[3] The parid argument to `DitsInit` can be retrieved by a user action routine using `DitsGetParid`

[4] This command is normally defined in the system wide login to be `@DRAMADISK:[000000]SYS_START_STOP DRAMASTART`, with `DRAMADISK:` defined in the system wide startup to point to the **DRAMA** directory structure.

[5] This assumes a dummy user account named `drama` has been setup to contain the **DRAMA** directory structure. It also assumes you are running a shell which supports the tilder format.

The ADITS program is started when you execute the command DITS_NETSTART, after executing the DRAMASTART command. There should be only one copy of ADITS for each VMS group wishing to access Dits, but if ADITS has already been started by another user in your group, you will require Group Privilege to access it.

ADAM tasks should specify the name of Dits tasks in the following form - "*nodename*!!*taskname*", where *nodename* is the internet address of the node on which the Dits task is running and *taskname* is the Dits task name.

Beware that *nodename* is just somewhere to look for the Dits task. If a Dits task of the given name is already known to the local machine, then no check is made to ensure it resides on the specified machine. If you don't specify *nodename*, then the local machine is assumed.

As ADAM uses upper-case only for action, parameter and task names, Dits tasks which are to be accessible to ADAM should be sure to use only uppercase case for such names.

Value strings supplied in ADAM messages are converted to Dits arguments. In the conversion, each word in the value string (words are separated by spaces) is considered a separate argument value. The words are given the argument names `Argument`n where n is the word number, starting at one. Such arguments can be examined using the `Arg package`.

Arguments returned in messages from Dits tasks to ADAM tasks are converted to a string format. Each Sds value is converted into a string and separated from other values by a space. Sds arrays cannot be handled.

An ADAM Obey with name MONITOR is converted to a Dits Monitor Message type, allowing ADAM tasks to monitor the parameters of a Dits task. The first argument to the message should be the name of the Monitor Message (START/FORWARD/ADD/DELETE/CANCEL) with the message specific arguments following.

Since ADAM tasks cannot handle MONITOR messages themselves, a Dits task which sends a monitor message to an ADAM task will get an error message. For CONTROL messages of type "MESSAGE", which are intended to translate error codes, the ADITS program will attempt the translation and send the response back to the Dits task. Some versions of ADAM (2.0.2 onwards) accept CONTROL message of type "DEFAULT", so these may or may not work.

8 Compiling and Linking Dits tasks

It is possible to imagine various ways of building Dits tasks. They can be built under UNIX or VMS, as simple tasks using little more then the Dits routines themselves or as complex tasks which must be linked with a large number of library routines. We have tried to provide techniques which allow the building of the most complex case, while making the building of simple tasks simple.

Unfortunately, the major differences in the user interface to the various operating systems make it impossible to design a similar interface on each of the systems supported.

The best approach is to use the technique of *dmakefiles* described in [7]. The remainder of this section documents the basic techniques which [7] builds on.

We assume here the software organisation described in [6] for both VMS and UNIX machines.

8.1 Building under VMS

Before building Dits tasks under VMS, you must execute the `DRAMASTART` command.

Compilation time - include files

The Dits include files can now be found in `DRAMA_INCLUDE:`. By using an appropriate command line to the C compiler[6], you can specify them using simple double quote notation, such as-

```
#include "DitsFix.h"
```

You should avoid using the following style

```
#include "DRAMA_INCLUDE:DITSFIX.H"
```

This style is not portable on two counts. First, logical names are only available on VMS. Second, on UNIX machines, case is significant. Use include file names exactly as specified in the routine specifications in the appendices to this document.

Link time

There are two link techniques available. The `DITS_LINK` command is the simplest of the two. The first parameter to the `DITS_LINK` command is any valid VMS LINK parameter. The text up to the first comma or slash is taken to be the name of the program being created. The application routine can be a library: use the APPLIC/INC=APPLIC/LIB syntax or something similar.

The second parameter is by default null and can be any valid VMS LINK qualifiers, *including the preceding slash.*

The following example will link the Dits task MYTASK, against the library MYLIB, with the debugger enabled-

```
DITS_LINK MYTASK.OBJ,MYLIB/LIB /DEBUG/TRACE
```

Note that the DITS_LINK procedure takes two extra parameters. These are to assist in debugging Dits itself so are not documented here. See `DITS_DIR:DITS_LINK.COM` for more details.

As an alternative to the `DITS_LINK` command, you can specify a link options file on your own link command. This style of link is more flexible but more complex in the simple case. For the above example, you could use-

```
LINK/DEBUG/TRACE MYTASK.OBJ,MYLINK/LIB,DITS_DIR:DITS/OPT
```

[6]See the `/INCLUDE=` qualifier to the VMS C compiler.

8.2 Building under UNIX

Before building Dits tasks under UNIX, you must execute the local DRAMA development startup. This is done with the command `~drama/dramastart`.

Compilation time - include files

The Dits include files can now be found using by using the argument '$DITS_LIB/dits_cc' to the C compiler. In your source code, you should specify them using simple double quote notation, such as-

```
#include "DitsFix.h"
```

The quotes surrounding **$DITS_LIB/dits_cc** are grave accents (ascii code 60 hex). [7]

Link time

To link with the Dits libraries, include '$DITS_LIB/dits_link' on the compiler command line. For example, to compile and link mytask.c using the Gnu C compiler -

```
gcc -o mytask mytask.c `\$DITS_LIB/dits_cc` `$DITS_LIB/dits_link`
```

Note **`$DITS_LIB/dits_cc`** is not required if all you are doing is compiling to `.o` files.

8.3 Building under VxWorks

Before building Dits tasks for VxWorks, you must execute the local DRAMA development startup on the development machine (The Sun). This is done with the command "`~drama/dramastart vw68k`" (For 680x0 based VxWorks machines).

Compilation time - include files

The Dits include files can now be found using by using the argument '$DITS_LIB/dits_cc' to the C compiler. In your source code, you should specify them using simple double quote notation, such as-

```
#include "DitsFix.h"
```

The quotes surrounding **$DITS_LIB/dits_cc** are grave accents (ascii code 60 hex).

Dits include files also include various VxWorks system include files, so these should be accessible at compile time. [8]

[7] The construct **`$DITS_LIB/dits_cc`** results in the command **$DITS_LIB/dits_cc** being executed and the output from this command replacing it on the command line. **dits_cc** is a shell script containing an echo command which outputs the include file directories to search etc. needed to compile a Dits program.

[8] The file `drama/makefile.vw68k` is a prototype include file for AAO systems. It adds the necessary include file directory specifications to find the VxWorks system include files.

Link and Run time

Once you have your basic VxWorks object file, then you have two of possible options -

- The first is that you want to create an object module containing a complete task. i.e., all the required modules, other then the VxWorks run time library, which are required to run your program. You can do this by including '$DITS_LIB/dits_link' on the compiler command line. For example, to link mytask.c using the Gnu C compiler -

  ```
  gcc -o mytask mytask.o '$DITS_LIB/dits_link'
  ```

  Note that the quotes surrounding **$DITS_LIB/dits_link** are grave accents (ascii code 60 hex).

- The problem with the previous technique occurs when you wish to load multiple programs from different files, such as during testing. In this case, it is better not to link with the Dits library, but to instead load it separately. You can then load multiple programs using Dits, but only load one copy of the Dits library.

  Currently the way of doing this is a bit complex and is site specific. We assume you are logged onto the AAO VxWorks development Sun. After having done the "`~drama-/dramastart vw68k`" command, enter "`echo $DITS_LIB`". This will write a string to the terminal, something like-

  ```
  /home/aaossc/drama/dits/T2/vw68k
  ```

  Find the directory just before the final **/vw68k**, in this case **T2**. Now, on the VxWorks machine, execute the command-

  ```
  ld </home/aaossc/vw/drama/dits/T2/libdits.o
  ```

  Except replace **T2** with what you found in its place in the result of the echo command. You can now load your own object modules.

9 User Interface Construction

One interesting thing about Dits that almost everything takes place in the context of an `Action`. This raises the question of just how we get things going - how is the first action started.

9.1 Uface context

The routine **DitsUfaceCtxEnable()** is the key. This routine can only be called outside of an action routine, for example, in the main loop or in a response routine specified by it. Its purpose is to enable a subset of Dits routines allowing a user interface to initiate and respond to Dits messages. The call to this routine is

```
DitsUfaceCtxEnable(DitsActionRoutineType ResponseRoutine
                   long int code,
                   StatusType *status)
```

After this routine is called, a special context, **DITS_CTX_UFACE** is current. The following routines will be available-

Routine	Function
`DitsObey`	Send an obey message
`DitsKick`	Send a kick message
`DitsGetPath`	Get a path to a task
`DitsGetParam`	Get the value of a parameter
`DitsSetParam`	Set the value of a parameter
`DitsGetParid`	Get the parid value supplied to DitsPutActionHandlers
`DitsGetContext`	Will return **DITS_CTX_UFACE**
`DitsInitiateMessage`	Send a message of a specified type.

Most of these routines start transactions, which normally result in the rescheduling of an action. When messages about transactions started under the UFACE context are received, a call will be made to the **ResponseRoutine** specified in the preceding call to **DitsUfaceCtxEnable()**.

The **ResponseRoutine** will also be called with a context of **DITS_CTX_UFACE** and in addition to the routines listed above, the following routines will be available-

Routine	Function
`DitsGetReason`	Get the reason for the entry.
`DitsGetCode`	Get the code supplied in the call to **DitsUfaceCtxEnable()**.
`DitsGetArgument`	Get the argument associated with the incoming message.
`DitsGetExtReason`	Get full details of the incoming message.
`DitsPutRequest`	The only applicable request is **DITS_REA_EXIT** to force the task to exit.

You may also want to call **DitsUfaceCtxEnable()** in the response routine to change the handler for messages initiated by the response routine.

`UFACE` context should be re-enabled after each all to **DitsMsgReceive()** otherwise the above routines will be undefined.

The user interface context cannot chose to ignore messages, you cannot call DitsNotInterested on their behalf. All messages resulting from the transaction will be delivered to the response routine.

A few other routines with names starting with `DitsUface` provide addition support for user interface context. In addition the **Dui** package wraps up the details in friendly way -

10 Current status

The current release is R0.15, available under Vax/Vms, Sun OS 4.1.x, Sun Os 5.4 (Solaris 2.4), Alpha/Osf 3.x, VxWorks 5.1.x.

The following still needs to be done-

- Support logging.

A Procedure Types

Several **Dits** routines and data structures take procedure arguments. In order to supply routines of appropriate types, you need to know how such routines are defined. This section gives the C typedef of all the routine types.

- ```
  typedef DVOID (*DitsActionCleanupRoutineType)(
                long int code,          /* Code assocaited with the action */
                StatusType *status);
  ```

  Defined in DitsTypes.h. See **DitsPutActions()** for details.

- ```
  typedef void (*DitsActionRoutineType)(StatusType *status);
  ```

  Defined in DitsTypes.h. See **DitsPutActions()** for details.

- ```
  typedef int  (*DitsConnectRoutineType)(
                void * client_data,
                const char * taskname,/* (>) Name of the connecting task */
                DitsConnectInfoType *ConnectInfo, /* Buffer size details */
                int  * flags,         /* (!) Connection flags            */
                StatusType *status);
  ```

  Returns true to indicate the connection should be accepted and false to reject it. Set the flag `DITS_M_FLOW_CONTROL` to indicate that connections to remote tasks should be flow controlled. See the Imp documentation for details of flow control.

  Defined in DitsSys.h. See **DitsPutConnectHandler()** for details.

- ```
  typedef void (*DitsDefaultRoutineType)(
                  void *client_data,
                  const char * new,      /* New directory name           */
                  int    resultlen,      /* Space in result string       */
                  char * result,         /* The resuling directory spec  */
                  StatusType *status);
  ```

  Defined in DitsFix.h. See **DitsPutDefaultHandler()** for details.

- ```
  typedef void  (*DitsDisConnectRoutineType)(
                 void * client_data,
                 const char * taskname,/* (>) Name of the task           */
  ```
  ```

```
                DitsPathType path,    /* (>)The path involved           */
                StatusType *status);
```

Defined in DitsSys.h. See **DitsPutDisConnectHandler()** for details.

- ```
  typedef void (*DitsGetRoutineType)(
              void * parid,             /* (>) Parameter system id       */
              char * parname,           /* (>) Parameter name            */
              DitsArgType * argument,   /* (<) Value written here        */
              int * delete,             /* (<) Set true if argument can  */
                                                  /*  be deleted       */
              StatusType *status);
  ```

  Defined in DitsParam.h. See **DitsAppParamSys()** for details.

- ```
  typedef void (*DitsInputCallbackRoutineType)(
               void *client_data,
               StatusType *status);
  ```

```
typedef void (*DitsMGetRoutineType)(
              void * parid,             /* (>) Parameter system id       */
              char * names,             /* (>) Parameter names           */
              DitsArgType * argument,   /* (<) Value written here        */
              int * delete,             /* (<) Set true if argument can  */
                                                  /*  be deleted       */
              StatusType *status);
```

Defined in DitsParam.h. See **DitsAppParamSys()** for details.

- ```
  typedef void (*DitsMsgOutRoutineType)(
               char * string,           /* (>) String to be output       */
               void * client_data,
               StatusType *status);
  ```

  Defined in DitsSys.h. See **DitsUfacePutMsgOut()** for details.

- ```
  typedef void (*DitsMonitorCheckRoutineType)(
               const char *name,      /* (>) Name of parameter    */
               StatusType *status);
  ```

  Defined in DitsParam.h. See **DitsAppParamSys()** for details.

- ```
      typedef void (*DitsMonitorDisconRoutineType)(
                 Dits___PathType *path,
                 StatusType *status);
  ```

  Defined in DitsParam.h. See **DitsAppParamSys()** for details.
  ```

- ```
  typedef DVOID (*DitsMonitorGetRoutineType)(
                const char *name,    /* (>) Name of parameter     */
                SdsIdType * paramId, /* (<) Value returned to here */
                StatusType *status);
  ```

  Defined in DitsParam.h. See **DitsAppParamSys()** for details.

- ```
  typedef void (*DitsMonitorMsgRoutineType)(
          INT32 flags,
                char *name,
                DitsArgType argin,
                Dits___NetTransIdType *transid,
                Dits___PathType *path,
                int *complete,
  ```

  Defined in DitsParam.h. See **DitsAppParamSys()** for details.

- ```
  typedef unsigned long int (*DitsMonitorSizeRoutineType)(
                const char *name,   /* (>) Name of parameter    */
                StatusType * status);
  ```

- ```
      typedef void (*DitsMonitorTidyRoutineType)(StatusType *status);
  ```

  Defined in DitsParam.h. See **DitsAppParamSys()** for details.

- ```
  typedef void (*DitsSetRoutineType)(
               void * parid,             /* (>) Parameter system id       */
               char * parname,           /* (>) Parameter name            */
               DitsArgType * argument,   /* (>) The new value             */
               StatusType *status);
  ```

  Defined in DitsParam.h. See **DitsAppParamSys()** for details.

- ```
  typedef void (*DitsSpawnCheckRoutineType)(
               void * client_data,
               StatusType *status);
  ```

  Defined in DitsTypes.h. See **DitsPutActions()** for details.

- ```
  typedef DVOID (*DitsWaitRoutineType)(
               void * client_data,
               StatusType *status);
  ```

  Defined in DitsSys.h. See **DitsPutEventWaitHandler()** for details.

- ```
  typedef void (*DuiCompleteHandlerType)(
               DuiDetailsType *Details, /* Details of the transaction  */
               StatusType *status);
  ```

  Defined in Dui.h. See **DitsExecuteCmd()** for details.

- ```
  typedef void (*DuiCommandCallbackType)(
                char *CommandString,     /* Use entered command */
                void * client_data,
                StatusType *status);
  ```

  Defined in Dui.h. See **DitsExecuteCmd()** for details.

- ```
  typedef void (*DuiLoadCompHandlerType)(
                DuiLoadDetailsType *details, /* Load operation details */
                StatusType *status);
  ```

  Defined in Dui.h. See **DitsLoad()** for details.

- ```
  typedef void (*DuiLoadHandlerType)(
                DuiLoadDetailsType *details, /* Load operation details */
                StatusType *status);
  ```

  Returns true to indicate the message has been handled. A False tell Dui to handle the message.

  Defined in Dui.h. See **DitsLoad()** for details.

- ```
  typedef int (*DuiHandlerType)(
                DuiDetailsType *Details, /* Details of the transaction  */
                StatusType *status);
  ```

  Returns true to indicate the message has been handled. A False tell Dui to handle the message.

  Defined in Dui.h. See **DitsExecuteCmd()** for details.

B SDS Structure Types

Messages sent between tasks are sent as SDS structures. The application programmer can add his own SDS data structures to messages which the receiver access using DitsGetArgument. SDS is by definition Self-defining, but in some cases it helps if the format of the Data sent is defined externally. This ensures it can be handled completely and efficiently. As a result, **Dits** defines SDS structures with the following names-

B.1 ArgStructure

Argument structures are normally built and interpreted by the **Arg** routines, defined in [3]. Argument structures consist of a set of scaler or string items. A string item is defined as a character array containing a null terminated string. Normally, argument structures do not contain other structures, since these cannot be interpreted by the **Arg** routines.

B.2 ImageStructure

An Image Structure contains a two dimensional array of scaler items which is to be interpreted as an image. User interfaces may display such images automatically when they are received.

The item **DATA_ARRAY** contains the actual data array, which can be of any scaler type.

The optional item **OFFSETS**, if supplied, indicates the offset of the image window within the detector window. It is a two element one dimensional array. The first element is the offset on the X direction, the second the offset in the Y direction. This item is normally only used by user interfaces dealing with detector systems. It allows them to set window's etc. based on points in the image.

B.3 MsgStructure

A Msg Structure is used to pass information messages to the user interface (Created by MsgOut calls). They contain the following items:

TASKNAME A character string containing the name of the task originating the message

MESSAGE A character string containing the actual message.

B.4 ErsStructure

An Ers Structure is used to pass error messages to the user interface (Created by Ers calls). They contain the following items:

TASKNAME A character string containing the name of the task originating the message

MESSAGE An array of character strings containing the actual messages.

FLAGS An array of integers containing the Ers flags passed in calls to ErsRep and ErsOut.

STATUS An array of integers containing the Status supplied to calls to ErsRep and ErsOut.

C Detailed Subroutine Descriptions

The following are details of all the Dits routines which have been implemented at this stage.

C.1 DitsActIndexByName — Returns the action index of an action of a specified name.

Function: Returns the action index of an action of a specified name.

Note, we only match the last entry of the name and ignore spawned actions.

Description:

Language: C

Call:
(long int) = DitsActIndexByName (name, index, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) name (char `*`) The name of the action to look up.

(`<`) index (long int `*`) The index to the action is returned here. If supplied as null, then the value is returned only.

(`!`) status (StatusType `*`) Modified status.

Return Value: The action index or -1 on error.

Include files: DitsUtil.h

External functions used:

External values used: DitsTask

Prior requirements: Can only be called from a Dits Application routine

See Also: The Dits Specification Document, DitsKillByIndex(3), DitsSignalByName(3), DitsGetActIndex(3)

Support: Tony Farrell, `AAO`

C.2 DitsActionTransIdWait — Blocks the current action and waits for a message to be received.

Function: Blocks the current action and waits for a message to be received.

Description: The current action is blocked and a message receive loop entered. Other actions can be processed as normal. When a message concerning this action is received, the action is unblocked and will continue. The current entry details (DitsGetArgument() DitsGetEntInfo() etc) will then be for the first message received for this action. The count variable will return the number of other messages still remaining to be processed.

The outstanding messages can be processed by calling this routine repeatly. The action (not the task) will block if the last call returned a count of 0, except if the `DONT_BLOCK` flag is set.

Note that if after this call is invoke, another action is invoked and calls DitsActionWait/DitsActionTransIdWait/DitsUfaceWait/DitsUfaceTransIdWait, then second action must be unblocked and end/reschedule before control will return to the first action.

If messages received while the action was blocked have not been processed when the action completes, they are lost (a message is output to stderr). To ensure these messages are

handled, you should continue calling this routine until count is returned as 0 or reschedule and handle them in the normal reschedule event process.

The action is `NOT` unblocked when a `KICK` is received for the action which invokes it, but the Kick handler will be invoked may cause the action to be rescheduled and the action will then be unblocked when the reschedule occurs. An except to this is if the `KICK` handler causes the action to end. In that case, this routine will return with either the status of the `KICK` handler (if not `STATUS__OK`) or with the a `STATUS` of `DITS__WAIT_ABORTED`.

If the transaction id argument is non-zero, then will only return if that transaction completes.

This routine may be invoked from either Obey or Kick context of an action, but not both at the same time (status = `DITS__WAITALRDY` will be returned). If invoked from a Kick handler, the obey handler may stage. If the obey handler causes the action to end, then this routine will return with either the status of the Obey handler (if not `STATUS__OK`) or with the a `STATUS` of `DITS__WAIT_ABORTED`.

If an action which has invoked this routine is killed using DitsKillByName(3)/DitsKillByIndex(3), then this routine will return with a status of `DITS__WAIT_ACT_KILLED` or the status supplied to the kill routine. The caller should return immediately - whist it could ignore this bad status and cause an allow the action to continue - this is considered rude as the kill routines should cause the action to die.

If after doing a call to DitsActionWait() which returned with `count>0`, you then reschedule your action (using DitsPutRequest()) with a request of `DITS_REQ_STAGE`, `DITS_REQ_WAIT`, `DITS_REQ_MESSAGE` or `DITS_REQ_SLEEP`, then `DRAMA` will commence processing your list of outstanding transactions, as normal reschedule events - i.e. your obey handler is invoked for each message on the list.

During this processing:

1. You are free to send other messages.

2. If you want to continue processing items from the list in this fashion - you must call DitsPutRequest() with a request of `DITS_REQ_STAGE`, `DITS_REQ_WAIT`, `DITS_REQ_MESSAGE` or `DITS_REQ_SLEEP` before returning.

3. You are free to call DitsActionWait() or DitsActionTransIdWait(). These calls will process the next relevant message on the list and can cause the list of messages to be processed.

4. If your handler returns with bad status, or a request of `DITS_REQ_END` (the default) or `DITS_REQ_EXIT`, then the action ends immediately without processing the rest of the list. The messages are lost (with a warning output to stderr).

5. The only way to process any new messages for your action/task during this processing is to call DitsActionWait() or DitsActionTransIdWait(). Otherwise they will be processed when processing of the list completes.

6. If the reschedule that started all this requested a delay triggered entry to your action (i.e. specified a delay with a request is DITS_REQ_MESSAGE/DITS_REQ_SLEEP or specified DITS_REQ_WAIT or DITS_REA_STAGE) then that delay is set up after all items on the list have been processed - including new items added to the list by calls to DitsActionWait() or DitsActionTransIdWait().

7. If a kick message for your action arrives whilst processing this list, presuming you have called DitsActionWait() or DitsActionTransIdWait(), it will be delivered. But any attempt by the kick message to cause the obey to reschedule (say the kick does a DitsPutRequest() with DITS_REQ_STAGE or DITS_REQ_WAIT, or it invokes DitsSignal() to itself) will result in a new message at the END of the list. It won't be proceessed until all other messages on the list are processed. If the kick causes the action to end at this point, the behaviour is the normal behaviour for DitsActionWait() or DitsActionTransIdWait() (see above).

Language: C

Call:
(void) = DitsActionTransIdWait (flags, timeout, transId, count, status)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(>) flags (int) A bit mask of flags. The possiblities are any or-ed combination of-

DITS_M_AW_FORGET	Forget any remaining messages from previous calls to this routine for this action.
DITS_M_AW_DONT_BLOCK	Always return immediatley. We will only read a message if there was one outstanding from a previous call. If there are no outstanding messages or DITS_M_FORGET flag has been set, then no message will be read and count will be set to -1. Action details, such as DitsGetArgument etc., will then be as per prior to the call.
DITS_M_AW_NO_LIST	Don't take an entry from the list. If combined with DONT_BLOCK we simply get the count of outstanding messages. Otherwise we block waiting for a message. When the call returns, it will be as per the first item on the list. It is not clear if this flag is usefull to users without the DONT_BLOCK flag.

(>) timeout (DitsDeltaTimeType *) If non-zero, the address of a reschedule delay.

(>) transId (DitsTransIdType) IF non-zero, the ID of a transaction to wait for. This must be a transaction ID of a message sent (or to be delivered to) this action.

(<) count (int *) Set to the number of messages remaining to be processed. The message which caused the routine to be unblocked is not included. If the `DITS_M_DONT_BLOCK` flag was set, then count will be -1 if no messages were available. Count is always non-negative in other cases.

(!) status (StatusType *) Modified status. `WARNING` - status of message receiving code, not the message status. To get the status of the received message, fetch the value from DitsGetEntStatus(). Also - a timeout does not set the status bad, it sets DitsGetEntReason() to `DITS_REA_RESCHED`.

Include files: DitsInteraction.h

External functions used:

ImpClearReminder	Imp	Clear outstanding reminders
ImpQueueReminder	Imp	Queue reminders
ImpReadEnd	Imp	Indicate completion of a read by pointer
SdsDelete	Sds	Delete a structure
SdsFreeId	Sds	Free a id
DitsMsgReceive	Dits	Received a dits message.

External values used: DitsTask - Details of the current task

Prior requirements: Can only be invoked in an action's obey or kick context.

See Also: The Dits Specification Document, DitsPutEventWaitHandler(3), DitsUfaceDits(3).

Support: Tony Farrell, `AAO`

C.3 DitsActionWait — Blocks the current action and waits for a message to be received.

Function: Blocks the current action and waits for a message to be received.

Description: The current action is blocked and a message receive loop entered. Other actions can be processed as normal. When a message concerning this action is received, the action is unblocked and will continue. The current entry details (DitsGetArgument() DitsGetEntInfo() etc) will then be for the first message received for this action. The count variable will return the number of other messages still remaining to be processed.

The outstanding messages can be processed by calling this routine repeatly. The action (not the task) will block if the last call returned a count of 0, except if the `DONT_BLOCK` flag is set.

Note that if after this call is invoke, another action is invoked and calls DitsActionWait/DitsActionTransIdWait/DitsUfaceWait/DitsUfaceTransIdWait, then second action must be unblocked and end/reschedule before control will return to the first action.

If messages received while the action was blocked have not been processed when the action completes, they are lost (a message is output to stderr). To ensure these messages are handled, you should continue calling this routine until count is returned as 0 or reschedule and handle them in the normal reschedule event process.

The action is `NOT` unblocked when a `KICK` is received for the action which invokes it, but the Kick handler will be invoked may cause the action to be rescheduled and the action will then be unblocked when the reschedule occurs. An except to this is if the `KICK` handler causes the action to end. In that case, this routine will return with either the status of the `KICK` handler (if not `STATUS__OK`) or with the a `STATUS` of `DITS__WAIT_ABORTED`.

This routine may be invoked from either Obey or Kick context of an action, but not both at the same time (status = `DITS__WAITALRDY` will be returned). If invoked from a Kick handler, the obey handler may stage. If the obey handler causes the action to end, then this routine will return with either the status of the Obey handler (if not `STATUS__OK`) or with the a `STATUS` of `DITS__WAIT_ABORTED`.

If an action which has invoked this routine is killed using DitsKillByName(3)/DitsKillByIndex(3), then this routine will return with a status of `DITS__WAIT_ACT_KILLED` or the status supplied to the kill routine. The caller should return immediately - whist it could ignore this bad status and cause an allow the action to continue - this is considered rude as the kill routines should cause the action to die.

If after doing a call to DitsActionWait() which returned with `count>0`, you then reschedule your action (using DitsPutRequest()) with a request of `DITS_REQ_STAGE`, `DITS_REQ_WAIT`, `DITS_REQ_MESSAGE` or `DITS_REQ_SLEEP`, then `DRAMA` will commence processing your list of outstanding transactions, as normal reschedule events - i.e. your obey handler is invoked for each message on the list.

During this processing:

1. You are free to send other messages.

2. If you want to continue processing items from the list in this fashion - you must call DitsPutRequest() with a request of `DITS_REQ_STAGE`, `DITS_REQ_WAIT`, `DITS_REQ_MESSAGE` or `DITS_REQ_SLEEP` before returning.

3. You are free to call DitsActionWait() or DitsActionTransIdWait(). These calls will process the next relevant message on the list and can cause the list of messages to be processed.

4. If your handler returns with bad status, or a request of `DITS_REQ_END` (the default) or `DITS_REQ_EXIT`, then the action ends immediately without processing the rest of the list. The messages are lost (with a warning output to stderr).

5. The only way to process any new messages for your action/task during this processing is to call DitsActionWait() or DitsActionTransIdWait(). Otherwise they will be processed when processing of the list completes.

6. If the reschedule that started all this requested a delay triggered entry to your action (i.e. specified a delay with a request is DITS_REQ_MESSAGE/DITS_REQ_SLEEP or specified DITS_REQ_WAIT or DITS_REA_STAGE) then that delay is set up after all items on the list have been processed - including new items added to the list by calls to DitsActionWait() or DitsActionTransIdWait().

7. If a kick message for your action arrives whilst processing this list, presuming you have called DitsActionWait() or DitsActionTransIdWait(), it will be delivered. But any attempt by the kick message to cause the obey to reschedule (say the kick does a DitsPutRequest() with DITS_REQ_STAGE or DITS_REQ_WAIT, or it invokes DitsSignal() to itself) will result in a new message at the END of the list. It won't be processed until all other messages on the list are processed. If the kick causes the action to end at this point, the behavior is the normal behavior for DitsActionWait() or DitsActionTransIdWait() (see above).

Language: C

Call:
(void) = DitsActionWait (flags, timeout, count, status)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(>) flags (int) A bit mask of flags. The possibilities are any or-ed combination of-

DITS_M_AW_FORGET	Forget any remaining messages from previous calls to this routine for this action.
DITS_M_AW_DONT_BLOCK	Always return immediatley. We will only read a message if there was one outstanding from a previous call. If there are no outstanding messages or DITS_M_FORGET flag has been set, then no message will be read and count will be set to -1. Action details, such as DitsGetArgument etc., will then be as per prior to the call.
DITS_M_AW_NO_LIST	Don't take an entry from the list. If combined with DONT_BLOCK we simply get the count of outstanding messages. Otherwise we block waiting for a message. When the call returns, it will be as per the first item on the list. It is not clear if this flag is usefull to users without the DONT_BLOCK flag.

(>) timeout (DitsDeltaTimeType *) If non-zero, the address of a reschedule delay.

(<) count (int *) Set to the number of messages remaining to be processed. The message which caused the routine to be unblocked is not included. If the DITS_M_DONT_BLOCK

flag was set, then count will be -1 if no messages were available. Count is always non-negative in other cases.

(!) **status (StatusType *)** Modified status. `WARNING` - status of message receiving code, not message status. To get the status of the received message, fetch the value from DitsGetEntStatus(). Also - a timeout does not set the status bad, it sets DitsGetEntReason() to `DITS_REA_RESCHED`.

Include files: DitsInteraction.h

External functions used:

ImpClearReminder	Imp	Clear outstanding reminders
ImpQueueReminder	Imp	Queue reminders
ImpReadEnd	Imp	Indicate completion of a read by pointer
SdsDelete	Sds	Delete a structure
SdsFreeId	Sds	Free a id
DitsMsgReceive	Dits	Received a dits message.

External values used: DitsTask - Details of the current task

Prior requirements: Can only be invoked in an action's obey or kick context.

See Also: The Dits Specification Document, DitsPutEventWaitHandler(3), DitsUfaceDits(3).

Support: Tony Farrell, `AAO`

C.4 DitsAltInAdd — Add an Alternative Input Source

Function: Add an Alternative Input Source

Description: The Alternative input source is recorded in the variable of type DitsAltInType for passing later to DitsAltInLoop

This routine can be called multiple times for the same input source to specify multiple routines to be called when the source event occurs.

The variable supplied to this routine should be cleared by calling DitsAltInClear() prior to the first call to this routine.

You can remove an entry by calling DitsAltInDelete(), specifing the same fd/event flag number, condition, proc and client data.

Multiple events may be specified for the same fd/event flag, as long as one of the condition/client_data/proc are different (condition is ignored under `VMS`).

A maximum of 40 alternative inputs can be added to one variable of type DitsAltInType.

Language: C

Call:

(Int) = DitsAltInAdd (v, source, condition, proc, client_data, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) v (DitsAltInType *) The variable of type DitsAltInType to set

(>) source (Long Int) Specifies the source file descriptor on a `UNIX`/VxWorks based system, Event flag on a `VMS` based system or other operating system-dependent device specification. (This will normally be the same source type as you would supply to XtAppAddInput()). The `VMS` event flag must be in cluster 0 only.

(>) condition (int) On `UNIX`/VxWorks systems this is some union or `DITS_M_READ_MASK`, `DITS_M_WRITE_MASK` and `DITS_M_EXCEPT_MASK` indicating the conditions we are interested in for the specified source. Under `VMS`, this is ignored.

Under `WIN32`, this is either a union of `DITS_M_READ_MASK`, `DITS_M_WRITE_MASK` or `DITS_M_EXCEPT_MASK`, indicating source is a socket. Or it is `DITS_M_WIN32_EVENT` indicating source is a `WIN32 HANDLE` we want to wait for an event on. Or it is `DITS_M_WIN32_MSG`. This last case indicates we should look for `WIN32` Message queue events. You can in this case construct "condition" using DitsWin32QueueMask(), which takes as its argument a mask of `WIN32` message events to look for. See the `WIN32` routine GetQueueStatus() for a list of the mask values. DitsWin32QueueMask returns an appropiate value for condition. If you just specify `DITS_M_WIN32_MSG`, then all `WIN32` message events will be examined.

(>) proc (DitsInputCallbackRoutineType) Specifies the procedure to be called when input is available on the above source.

(>) client_data (void *) Specifies the argument that is to be passed to the specified procdure when input is available

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used:

`FD_SET`	`UNIX`/VxWorks	Set a file descriptor mask element.
ErsRep	Ers	Report error messages.

External values used: none

Prior requirements: v must have been cleared by calling DitsAltInClear.

See Also: The Dits Specification Document, DitsAltInClear(3), DitsAltInLoop(3), DitsAltInDelete(3).

Support: Tony Farrell, `AAO`

C.5 DitsAltInClear — Clear an variable of type DitsAltInType

Function: Clear an variable of type DitsAltInType

Description: This routine sets up a variable of type DitsAltInType in prepration for calls to DitsAltInAdd and DitsAltInLoop.

Language: C

Call:
(Void) = DitsAltInClear (v, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) v (DitsAltInType *) The variable of type DitsAltInType to clear

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used:

`FD_ZERO`	Unix/VxWorks	Clear a file descriptor mask

External values used: none

Prior requirements: none

See Also: The Dits Specification Document, DitsAltInAdd(3), DitsAltInLoop(3), DitsAltInDelete(3)

Support: Tony Farrell, `AAO`

C.6 DitsAltInDelete — Delete an Alternative Input Source

Function: Delete an Alternative Input Source

Description: The Alternative input source is delete in the variable of type DitsAltInType

You must specify the same set of source/condition/proc/client_data as specified to DitsAltInAdd().

Language: C

Call:
(Int) = DitsAltInDelete (v, source, condition, proc, client_data, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) v (DitsAltInType *) The variable of type DitsAltInType to set

(>) source (Long Int) See DitsAltInAdd.

(>) condition (int) See DitsAltInAdd.

(>) proc (DitsInputCallbackRoutineType) See DitsAltInAdd.

(>) client_data (void *) See DitsAltInAdd

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used:

`FD_CLR`	`UNIX`/VxWorks	Set a file descriptor mask element.
ErsRep	Ers	Report error messages.

External values used: none

Prior requirements: v must have been cleared by calling DitsAltInClear.

See Also: The Dits Specification Document, DitsAltInClear(3), DitsAltInLoop(3), DitsAltInAdd(3).

Support: Tony Farrell, `AAO`

C.7 DitsAltInLoop — Implements a Dits Main loop with alternative sources of message events.

Function: Implements a Dits Main loop with alternative sources of message events.

Description: The normal DitsMainLoop routine loops awaiting only Dits Message events. By using the `DITS_M_X_COMPATIBLE` flag to DitsAppInit, you can add Dits Messages to the list of input events for an X program.

This routine provides for another common requirement where the alternative source of inputs is not X windows, but a File.

You must call DitsAppInit with the flag `DITS_M_X_COMPATIBLE` enabled for this routine to work.

Alternative sources of input are setup using the DitsAltInAdd routine. Currenlty, 10 other sources are possible.

On `UNIX`/VxWorks systems, the alternative sources are normally file descriptors and the corresponding condition determines if we are interested in read/write/exception conditions. Under `UNIX`/VxWorks, this routine uses select() to wait for file/message system events.

On `VMS` systems, the alternative sources are Event flags. The routine uses `SYS$WFLOR` to wait for event flag/message system events.

Language: C

Call:

(Void) = DitsAltInLoop (v, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) v (DitsAltInType `*`) The variable of type DitsAltInType describing alternative sources of input.

(`!`) status (StatusType `*`) Modified status. Possible failure codes are anything from the Dits and `VMS` routine used and

`DITS__INVDESC`	Select() set errno to `EBADF`.
`DITS__SIGNAL`	Select() set errno to `EINTR`.
`DITS__SELECTERR`	Any other error from select.

Include files: DitsSys.h

External functions used:

DitsAltInSet	Dits	Set an alternative source of input
DitsGetXInfo	Dits	Get X compatibilty details
DitsMsgAvail	Dits	Get the count of outstanding messages
DitsMsgReceive	Dits	Receive and process dits messages
sys$wflor	`VMS`	Vms only, wait on the or of some event flags
sys$readef	`VMS`	Vms only, Read an event flag state
select	`UNIX`/VxWorks	Wait for events on a set of file descriptors.

External values used: none

Prior requirements: DitsAppInit() must have been called.

See Also: The Dits Specification Document, DitsAltInAdd(3), DitsAltInClear(3), DitsMainLoop(3), DtclAppMainLoop(3), DtclAppTkMainLoop(3)

Support: Tony Farrell, `AAO`

C.8 DitsAppInit — Initialise Dits.

Function: Initialise Dits.

Description: Allocate memory for the variable DitsTask, which maintains all information about the task. When running under VxWorks, we must add this to the Task context switch block as all tasks run in common memory.

Register this task as a `DRAMA` (Imp) task.

Under VMS an exit handler is setup to handle an unexpected program exit correctly.

Under Unix, we attempt to setup an exit handler by setting up traps for the signals (when available)

SIGINT	Interrupt.
SIGQUIT	quit.
SIGPIPE	Write on a pipe or socket with no one to read.
SIGHUP	Hangup.
SIGTERM	Software termination signal.
SIGABRT	used by abort
SIGFPE	floating point exception
SIGILL	illegal instruction
SIGSEGV	segmentation violation
SIGALRM	Alarm clock.
SIGBUS	bus error
SIGEMT	EMT instruction
SIGIOT	IOT Instruction
SIGSYS	bad argument to system call
SIGTRAP	trace trap (not reset when caught)
SIGXCPU	Cpu time limit expired.
SIGXFSZ	File size limit exceeded.

You may insert your own signal handlers for these signals but try to ensure your program calls DitsStop when it exits. The Dits signal handler will shutdown dits and calls exit(3) in a forked subprocess and invokes the systems default handler for the signal

Under VxWorks, if this is the first Dits task, then a delete hook is created to be called when the task is deleted. The hook is only added by the first Dits task. A list is maintained of all active Dits tasks and when a Dits task exits, the exit handler is called. The last Dits task to exit will delete the hook.

Message codes for the DITS, SDS, IMP ERS and ARG libraries are enabled using MessPutFacility.

The the environment variable/logical name DITS_LOG is defined with an integer value, this is used as the initial value of the internal debug flag.

Language: C

Call:

(Void) = DitsAppInit (taskname, bytes, flags, initInfo, status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) taskname (Char *) The name the task is to be known by.

(>) bytes (Int) This argument indicates how many bytes should be allocated to the message buffer

This value is not allowed to be less then `DITS_C_MINSIZE`, which is the size of an `OBEY` command which has a zero length argument. Note that you should consider a buffer size equivalent to the maximum message size multiplied by the number of such messages that may be queued at any one time.

Each time another task opens a path to this task, it allocates some the space in this task's buffer for messages from the it.

It it hoped that the underlying message protocol will allow some technique for changing this buffer size, but it will nearly always be more efficient to get it right at this point.

(>) flags (Int) A flag word containing some options. Set the following bits to enable the options.

DITS_M_NO_LOCAL_SDS	Don't use Sds for local messages.
DITS_M_X_COMPATIBLE	specifies that the notification mechanism used by the task must be one that can coexit with the X-windows system. X systems can use the XtAppAddInput procedure call to specify that there are other events in the system that need to be handled, and if this flag is set, a suitable message notification mechanism will be used for incomming messages. (Note that this may be slower than the default mechanism). The routine DitsGetXInfo() can be used to get the information about the mechanism required for XtAppAddInput. *** WARNING *** Under VxWorks, the use of this notification method is incompatiable with using any of the DitsSignal*() series of routines from either and ISR or another task. A warning will be issued by the DitsSignal*() serious of routines if you attempt this. (The problem is that the notification method may theoretically block for a short period - which cannot be allowed from an ISR and can cause a task reschedule event when called from another task - even when the calling task has called taskLock(). The occurance and its impact are somewhat random.)
DITS_M_MAY_LOAD	If a load operation fails for a task on the local machine due to the IMP network tasks not existing, then the task may complete the load itself.
DITS_M_NOEXHAND	Don't install the exit handler. If you specify this flag then you a responsible for calling DitsStop in the event of a signal which normally causes the program to exit immediately. This flag allows you to handle all signals yourself.
DITS_M_NO_FC_AC	Indicates the connections accepted by this task (when other tasks attempts to initiate connections with this task) should not use flow control. By default flow control is enabled. This flag is ignored if you put your own connect handler using DitsPutConnectHandler, when you can set it from you connect handler on a task specific basis. In addition, it is also ignored when local tasks connect to this task (when it is not relevant).
DITS_M_FC_IC	Indicates the connections to other tasks initiated by DitsGetPath should have flow control enabled. You can override this on a connection specific basic by using DitsPathGet instead of DitsGetPath.
DITS_M_SELF_BYTES	Indicates the selfBytes item in the initInfo structure has been set.
DITS_M_GP_OLD	Indicates the DitsGetPath should operate in the old mode of returned immediately if the path already exists or we are already getting a path to the same task. See the DITS_M_PG_IMMED flag of the new DitsPathGet call for details

`DITS_M_REGISTRAR` specifies that the task wants to be notified when any task registers successfully with the `IMP` system. Such a task will receive messages that result in the routine specified by

() DitsPutRegistrationHandler () being invoked.

(>) initInfo (DitsInitInfoType *) Various items, dependent on which flags are set. If none of the relevant flags are set, this item is ignored so you may specify an address of 0.

selfBytes	Used if `DITS_M_SELF_BYTES` flag is set. This Number of bytes for the self buffer. This buffer is used for signal messages and for other cases where we want to send messages to ourselves. It should be at least 2000 bytes.

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used:

Dits___ExHandDeclare	Dits internal	Setup exit handlers.
malloc	`CRTL`	Allocate memory
taskVarAdd	VxWorks	When running under VxWorks only, add a variable to the task's switch block.
ImpRegister	`IMP`	Register a task with the message system.
Dits___SdsIdCreateTap	Dits internal	Create an Sds id.
Dits___SdsIdCreateGsok	Dits internal	Create an Sds id.
MessPutFacility	Mess	Add a new message facility.
MessGetMsg	Mess	Get the message text for an error code.
ErsStart	Ers	Startup Ers.

External values used: DitsTask - Details of the current task

Prior requirements: Should be called before any other Dits calls.

See Also: The Dits Specification Document, DitsInit(3), DitsPutActions(3), Imp manual, Ers manual, Sds manual, Messgen manual.

Support: Tony Farrell, `AAO`

C.9 DitsAppParamSys — Get or put parameter system details.

Function: Get or put parameter system details.

Description: This routine allows you to get or change the parameter system details.

Variables of type DitsParamSysType are used and they have the following elements.

id	void *. Client data item passed to getRoutine, mGetRoutine and setRoutine. Can be retreived by calling DitsGetParId().
getRoutine	Get parameter routine of type DitsGetRoutineType. Invoked to return the value of the specified parameter.
mGetRoutine	Multiple get routine of type DitsMGetRoutineType. Invoked to return the values of a number of parameters. The names are passed in a space separated list.
setRoutine	Set parameter routine of type DitsSetRoutineType. Invoked to set the value of a parameter.
monMsgHandler	Monitor message handler routine of type DitsMonitorMsgRoutineType. Routine to be called when a monitor message is received.
monDisconHandler	Monitor disconnect handler of type DitsMonitorDisconRoutineType. Routine to be called when a task has disconnected.
monTidyHandler	Monitor tidyup routine of type DitsMonitorTidyRoutineType. Routine to be called when the task shuts down (from DitsStop).
monCheckHandler	Monitor check existance routine of type DitsMonitorCheckRoutineType. Optional routine which is called to check a parameter for which monitoring is requested is valid. Should return an invalid status if the name is not valid.
monSizeHandler	Monitor size routine of type DitsMonitorSizeRoutineType. This Routine is is called to return the size which will be needed to send a message containing the named parameter.
monGetHandler	Monitor get routine of type DitsMonitorGetRoutineType. Routine is is called to return the value of the named parameter by Sds id. This is only used on rare occasions.
sdsDelete	If true, then id represents the top level Sds id of the parameter system. DitsStop will call SdsDelete and SdsFree on this id. (This flag gets around a problem where by there is no shutdown call in the Sdp parameter system. Other parameter systems should set it to 0.

This routine can be used to set the parameter system or as an inquiry.

Language: C

Call:

(void) = DitsAppParamSys(new,old,status,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) new (DitsParamSysType `*`) If supplied, new parameter system details. If a null pointer is supplied, we don't attempt to change parameter system details.

(`<`) old (DitsParamSysType `*`) If supplied, we return the current parameter system details here. If not supplied, ignored.

(`!`) status (StatusType `*`) Modified status.

Function Prototypes Used: typedef `DVOID` (*DitsGetRoutineType)(`DVOIDP` parid, `DCONSTV` char `*` parname, SdsIdType `*` argument1, int *delete_flag, StatusType `*` status);

typedef `DVOID` (*DitsMGetRoutineType)(`DVOIDP` parid, char *names, SdsIdType `*` argument1, int *delete_flag, StatusType `*` status);

typedef `DVOID` (*DitsSetRoutineType)(`DVOIDP` parid, `DCONSTV` char `*` parname, SdsIdType `*` argument2, StatusType `*` status);

typedef `DVOID` (*DitsMonitorMsgRoutineType)(`DCONSTV` `INT32` flags, `DCONSTV` char *monname, SdsIdType argin, Dits___NetTransIdType *transid, Dits___PathType *path, long int tag, int *complete, StatusType `*` status);

typedef `DVOID` (*DitsMonitorDisconRoutineType)(Dits___PathType *path, StatusType `*` status);

typedef `DVOID` (*DitsMonitorTidyRoutineType)(StatusType `*` status);

typedef `DVOID` (*DitsMonitorCheckRoutineType)(`DCONSTV` char *parname, StatusType `*` status);

typedef unsigned long int (*DitsMonitorSizeRoutineType)(`DCONSTV` char *parname, StatusType `*` status);

typedef `DVOID` (*DitsMonitorGetRoutineType)(`DCONSTV` char *parname, SdsIdType `*` argument3, StatusType *status);

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) parid (void `*`) Contains a copy of the value passed in the id item of the DitsParamSysType structure.

(`>`) parname (const char `*`) The name of the parameter. Note that for the get and monitor messages, the names `_ALL_`, `_NAMES_` and `_LONG_` are reserved. For MGet messages, the name is a list of space separated parameter names.

The meaning of the parameter name is specific to the parameter system, but, for example, in the Sdp parameter system, the name is passed to SdsFindByPath(3) with the `SDS` id being the id item of the DitsParamSysType structure.

`_ALL_`	For Get or MGet messages, this requests that the entire parameter system be returned. In this case, an Sds structure with each item representing a parameter value is returned. For monitor messgeas, this indicates all parameter values are to be monitored.
`_NAMES_`	Requests that a list of names be returned. In this case a structure of type SdpNames is returned. This will contain one item, an array with each of the names in it. This only applies to Get and Mget messages.
`_LONG_`	Set messages only. If the name of the parameter is `_LONG_`, then the argument structure contains the actual parameter name as well as the value. The first item in the structure is the name and the second is the value. This allows a long parameter value name to be specified in a `SET` message. To specify long parameter names in Get messages - An MGet message can be used (the list of names need only one value.

(<) argument1 (SdsIdType *) A `SDS ID` of a new `SDS` structure is return. It is to contain the requested parameter values in a from such that scaler parameters can be accessed using the Arg series of routines.

(>) argument2 (const SdsIdType *) A pointer to an `SDS ID` containing the new value for a parameter.

(<) argument3 (SdsIdType *) The `SDS ID` of an `SDS` item who's name is the name of a parameter and who values is the value of that parameter. (Note - a flaw exists that there is no way of indicating if this value should be deleted by `DITS` when `DITS` is finished with it. The `SDS` id is passed to SdsFreeId(3) but not to SdsDelete(3).

(>) delete_id (int *) If set true, then `DITS` should delete the `SDS` structure (and free the `ID`). Otherwise `DITS` will leave it.

(>) monname (const char *) The type of monitor message. One of `START//FORWARD/ADD/DELETE/CAN` See DitsInitiateMessage(3) for more details.

(>) flags (`INT32`) Flags associated with the monitor message. See DitsInitiateMessage(3) for more detailss. Possible values are

DITS_M_REP_MON_LOSS	cause the reporting of monitor messages which are lost due to waiting for buffer empty notification messages to arrive. If not set, lost messages can be ignored.
DITS_M_SENDCUR	if a START/FORWARD/ADD monitor message and you want the current value of the parameter sent immediately.

(>) transid (const Dits___NetTransIdType *) A DITS message transaction id. This can be supplied to the Dits___SendTap() function when sending replies back to the originator of the monitor message.

(>) path (const Dits___PathType *) The DITS path to the originator of the monitor message or the disconnecting path.

(>) tag (long) The tag assoicated with the original message. This is required by Dits___SendTap() when repling to monitor messages.

(<) complete (int *) Set true to indicate if the transaction is complete. START and FORWARD transactions would normally not complete, ADD/DELETE/CANCEL would normally complete immediately.

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used: None

External values used: DitsTask

Prior requirements: DitsAppInit should have been called.

See Also: The Dits Specification Document, DitsAppInit(3), DitsInitiateMessage(3), SdpCreate(3), SdpGet(3), SdpSet(3).

Support: Tony Farrell, AAO

C.10 DitsArgIsBulk — Indicate if the argument to an action entry is bulk data.

Function: Indicate if the argument to an action entry is bulk data.

Description: This function returns true (non-zero) if the argument for this reschedule of the current action is a bulk data item.

If DitsArgIsBulk(3) returns true, then the routine DitsBulkArgInfo(3) can be used to get the shared memory details. Having called DitsBulkArgInfo(3), you must explicitly release the shared memory using DitsBulkArgRelease(3).

Note, calling DitsBulkArgInfo(3) results in this routine returning false.

Language: C

Call:
(int) = DitsArgIsBulk()

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output) none

Returned Value: Returns non-zero to indicate the argument to the action entry is a bulk data item.

Include files: DitsSendBulk.h

See Also: DitsBulkArgInfo(3), DitsBulkArgRelease(3), DitsDefineShared(3), DitsReleaseShared(3), DitsDefineSdsShared(3), DitsTriggerBulk(3), DitsInitMessBulk(3), SdpCreateBulk(3).

Support: Tony Farrell, `AAO`

C.11 DitsBulkArgInfo — Returns details about a bulk data shared memory argument

Function: Returns details about a bulk data shared memory argument

Description: If the argument to the action entry is a bulk data shared memory item (DitsArgIsBulk(3) has returned true) then this routine can be used to get details of the bulk data.

This routine is required if you want to maintain access to the bulk data shared memory after your action routine returns or if you want to access the raw data (which will be necessary if your data is not an `SDS` item). You become responsible for the bulk data yourself and must call DitsBulkArgRelease(3) when you are finished with it.

If this routine is invoked as part of an action receiving bulk data, then you must call DitsBulkArgRelease(3) before `BEFORE` the action completes (Your action may reschedule - but it must not complete). If you fail to do this, the sending task may not be notified correctly when you have released the shared memory. `DRAMA` will attempt to warn you of this - either by setting status to `DITS__BULKRELERR` after your action completes, or if status was already bad, appending an error report.

If the item is an `SDS` item, the `SDS` id returned by DitsGetArgument(3) will be free-ed when your action handler routine returns, regardless of if you have called DitsBulkArgRelease(3). If you want to maintain `SDS` access across action reschedule events, you should call SdsAccess(3) yourself, using the data address returned by this routine. In addition, in either case, the `SDS` id is invalidated but not free-ed when you call DitsBulkArgRelease(3).

The DitsArgIsBulk() routine will return false after calling this routine and a second attempt to call this routine will fail.

If you wish to forward this bulk data item to another task, the SharedMemInfo item of the ReportInfo argument can be used to access an item of type DitsSharedMemInfoType

which describes the shared memory assoicated w ith this bulk data argument. This item can be used with DitsInitMessBulk(3) and DitsTriggerBulk(3).

Language: C

Call:
(void) = DitsBulkArgInfo(ReportInfo,Address,Size,NotifyBytes,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(<) ReportInfo (DitsBulkReportInfoType *) This structure must be passed to DitsBulkArgRelease(3) when you are finished with the bulk data. It is also used by DitsBulkArgReport(3) when you wish to report on bulk data usage.

(<) Address (void **) The address of the bulk data will be written here.

(<) Size (unsigned long *) The size in bytes of the bulk data will be written here.

(<) NotifyBytes (unsigned long *) The requested notification size is written here. If non-zero, you are requested to notify the sender after processing each increment of this number of bytes by using DitsBulkArgReport(3).

(!) status (StatusType *) Modified status.

Include files: DitsBulk.h

See Also: DitsArgIsBulk(3), DitsBulkArgRelease(3), DitsBulkArgReport(3), DitsDefineShared(3), DitsReleaseShared(3), DitsDefineSdsShared(3), DitsTriggerBulk(3), DitsInitMessBulk(3), SdpCreateBulk(3).

Support: Tony Farrell, `AAO`

C.12 DitsBulkArgRelease — Notify `DRAMA` you are finished with bulk data.

Function: Notify `DRAMA` you are finished with bulk data.

Description: This routine must be invoked if you have accessed a bulk data argument using DitsBulkArgInfo(3). It indicates to the sender of the bulk data that you have finished with the data and releases the shared memory.

Note 1, where this bulk data was received by a `DRAMA` action (as against a `UFACE` handler routine) the bulk data must be released before the action completes otherwise the originator of the Obey may not be notified correctly. If this call is made after the action has completed, then status will be set to `DITS__BULRELERR` after the release is done and `ERS` will be used to report details indicating a programming error in the task.

Note 2, if is not considered a failure if the task which sent the bulk data has disappeared and this will be quietly ignored.

Language: C

Call:
(void) = DitsBulkArgRelease(ReportInfo,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) ReportInfo (DitsBulkReportInfoType *) As returned by DitsBulkArgInfo(3).

(!) status (StatusType *) Modified status.

Include files: DitsBulk.h

See Also: DitsBulkArgInfo(3), DitsBulkArgReport(3), DitsDefineShared(3), DitsReleaseShared(3), DitsDefineSdsShared(3), DitsTriggerBulk(3), DitsInitMessBulk(3), SdpCreateBulk(3).

Support: Tony Farrell, `AAO`

C.13 DitsBulkArgReport — Notify `DRAMA` of the number of bulk data bytes used.

Function: Notify `DRAMA` of the number of bulk data bytes used.

Description: This routine allows you to notify the sender of bulk data of the progress of your use of the bulk data. The sender receives an `DITS_REA_BULK_TRANSFERRED` entry indicates the number of bytes has been used and can fetch details using DitsGetEntBulkInfo(3).

Note, if is not considered a failure if the task which sent the bulk data has disappeared and this is quietly ignored.

Language: C

Call:
(void) = DitsBulkArgReport(bytes,ReportInfo,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) bytes (unsigned long) Number of bytes used so far.

(!) ReportInfo (DitsBulkReportInfoType *) As returned by DitsBulkArgInfo(3).

(!) status (StatusType *) Modified status.

Include files: DitsBulk.h

See Also: DitsBulkArgInfo(3), DitsBulkArgRelease(3), DitsDefineShared(3), DitsReleaseShared(3), DitsDefineSdsShared(3), DitsTriggerBulk(3), DitsInitMessBulk(3), SdpCreateBulk(3).

Support: Tony Farrell, `AAO`

C.14 DitsCheckTransactions — Checks if there are outstanding transactions for the current action.

Function: Checks if there are outstanding transactions for the current action.

Description: This routine checks if the current action should be expecting further messages in regard to transactions it has initiated. If there are any outstanding transactions which the action has initiated, then this routine returns the number of them

Language: C

Call:
(int) = DitsCheckTransactions()

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output) none

Include files: DitsInteraction.h

Function value: Count of outstanding transactions to this action.

External functions used: none

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine. Otherwise allways returns 0.

See Also: The Dits Specification Document, DitsInitiateMessage(3), DitsForget(3).

Support: Tony Farrell, `AAO`

C.15 DitsDefault — Set/Return the default directory.

Function: Set/Return the default directory.

Description: Set a new default directory (if desired) and return the resulting default directory.

This routine is normally supplied to DitsPutDefaultHandler or invoked by a routine supplied to DitsPutDefaultHandler.

Language: C

Call:
(void) = DitsDefault(client_data,new,resultlen,result,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) client_data (void *) As supplied to DitsPutDefaultHandler.

(>) new (char *) The requested new default directory. If a null string is supplied, don't attempt to change the directory.

(>) resultlen (int) The space available in the result buffer. Should probably be `MAXPATHLEN` under posix.

(<) result (char *) The resulting default directory is written here.

(!) status (StatusType) Modified status

Include File: DitsUtil.h

External functions used:

chdir	`CRTL`	Change the default directory
getcwd	`CRTL`	`VMS`, VxWorks 5.0, linux and `WIN32`, Solaris. Get the default directory
getwd	`CRTL`	All other cases Get the default directory.

External values used: None

Prior requirements: None

See Also: The Dits Specification Document, chdir(2), getwd(3), getcwd(3), DitsPutDefaultHandler(3).

Support: Tony Farrell, `AAO`

C.16 DitsDefineSdsShared — Defined a shared memory segment and export an `SDS` structure into it.

Function: Defined a shared memory segment and export an `SDS` structure into it.

Description: This routine wraps up the common sequence of creating a shared memory segement using DitsDefineShared() and exporting an `SDS` structure into it. The `SDS` id of the exported structure is returned.

See DitsDefineShared() for full details on shared memory.

Language: C

Call:

(void) = DitsDefineSdsShared (Template, Type, Name, Key, Create, Address, SharedMemInfo, `ID`, Status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) Template (SdsIdType) Template Sds structure. The `SDS` structure placed in the shared memory shall be created using this `SDS` id and the SdsExportDefined() function.

(>) Type (int) A code used to descibe the mapping type to be used. This must be one of the DITS_SHARE_xxxx codes, see DitsDefineShared(3).

(>) **Name (char *)** Used to identify the mapped section on some machines. On `UNIX`, for example, if mapped files are used, this is the file name in question. If Name is not actually used on the current system it should be passed as a null string.

(>) **Key (int)** On some systems, this is an alternative way of specifying the mapped section. For example, if System V shared memory is used under `UNIX`, this is the i dentifier for that shared memory.

(>) **Create (int)** If true, the memory section does not actually exist and this routine is to create it. If this is the case, the Address field of the SharedMemInfo structure will give the address at which the memory section has been mapped, if it is mapped successfully.

(!) **Address (void **)** On systems with global address spaces, eg. VxWorks, the address of the memory is enough to specify it completely - together with its size. In any case, if the section is already mapped, this argument must specify the address at which it is mapped. If the section is created by this routine, this is returned with the address at which the memory was mapped, which is why this is passed as the address of a void pointer and not just the pointer.

(<) **SharedMemInfo (DitsSharedMemInfoType *)** The structure filled by this routine to describe the shared memory section.

(<) `ID` **(SdsIdType *)** Id of exported Sds structure. The User should free this id when they are finished with it (probably when DitsReleaseShared() is invoked).

(!) **Status (StatusType *)** Modified status. will be returned.

Include files: DitsBulk.h

See Also: The `DITS` Specification Document, DitsReleaseShared(3), DitsDefineShared(3), SdpCreateShared(3), DitsInitMessBulk(3), DitsTriggerBulk(3).

Prior requirements: None.

Support: Tony Farrell, `AAO`

C.17 DitsDefineShared — Defines and optionally creates a shared memory section.

Function: Defines and optionally creates a shared memory section.

Description: Shared memory sections are used by `DITS` in the handling of bulk data transfers, generally initiated through a call to DitsInitMessBulk(3) or DitsTriggerBulk(3).

These shared memory sections are generally created by the calling routine (in a highly system-dependent way) and then have to be described to `DITS` so it can use them. For this purpose, `DITS` has a structure called a 'shared memory information structure' of type DitsSharedMemInfoType. This routine is a utility that can be used by a program that has created such a shared memory section in order to simplify the task of filling out the

DitsSharedMemInfoType structure. It has the optional feature that it can also be used to create the structure if it does not yet actually exist. It is also possible to use this routine in a system-independent way to create a temporary shared memory section of a specified size. This can be used by a program that has no particular interest in controlling the details of the shared memory used, but merely wants a shared memory section created.

The parameters for this routine are used quite differently in the different systems for which `DITS` is implemented, and these system dependent details are described in more detail below. Note that this routine should always be used to initialise an DitsSharedMemInfoType structure, since it will always initialise those elements of the structure defined as being for `DITS` internal use only, and whose use is not explicitly documented.

Language: C.

Call:

(void) = DitsDefineShared (Type, Name, Key, Size, Create, Address, SharedMemInfo, Status)

Parameters: ("`>`" input, "`<`" output, "`!`" modified)

(>) Type (int) A code used to descibe the mapping type to be used. This must be one of the DITS_SHARE_xxxx codes, see the below.

(>) Name (char *) Used to identify the mapped section on some machines. On `UNIX`, for example, if mapped files are used, this is the file name in question. If Name is not actually used on the current system it should be passed as a null string.

(>) Key (int) On some systems, this is an alternative way of specifying the mapped section. For example, if System V shared memory is used under `UNIX`, this is the identifier for that shared memory.

(>) Size (long) The size in bytes of the shared memory section.

(>) Create (int) If true, the memory section does not actually exist and this routine is to create it. If this is the case, the Address field of the SharedMemInfo structure will give the address at which the memory section has been mapped, if it is mapped successfully.

(!) Address (void **) On systems with global address spaces, eg. VxWorks, the address of the memory is enough to specify it completely - together with its size. In any case, if the section is already mapped, this argument must specify the address at which it is mapped. If the section is created by this routine, this is returned with the address at which the memory was mapped, which is why this is passed as the address of a void pointer and not just the pointer.

(<) SharedMemInfo (DitsSharedMemInfoType *) The structure filled by this routine to describe the shared memory section.

(!) Status (StatusType *) Modified status. will be returned.

External variables used: None.

Include Files: DitsBulk.h

See Also: The DITS Specification Document, DitsReleaseShared(3), DitsDefineSdsShared(3), SdpCreateShared(3), DitsInitMessBulk(3), DitsTriggerBulk(3).

Prior requirements: None.

Support: Tony Farrell, AAO.

System-independent use: If Type is set to DITS_SHARE_CREATE, then this call creates a temporary shared memory section of the size specified by Size, in some suitable form for the system it is running on. The caller has no control over the details of the shared memory section. In this case, the Name and Key arguments, and any preset value at the location specified by the Address argument are all ignored. A new section will be created and mapped and the address at which it was mapped will be returned via Address. The Create argument is also ignored, since it is assumed to be set true.

System-dependent details:

The following considerations apply when calling this routine on different systems, with Type set to some value other than DITS_SHARE_CREATE.

VMS: The mapped section is determined entirely by the Name field, which should be the name of a global page section. Type should be passed as DITS_SHARE_GBLSC. If Type is set to zero, then DITS_SHARE_GBLSC is assumed. Key and Address are ignored.

VxWorks: The mapped section is just a section of memory, starting at a specified address. Type should be DITS_SHARE_GLOBAL. Name should be a Null string, and Key is ignored. If Create is specified, Address is ignored, and a suitably sized area of memory is allocated. If Create is passed as false, then Address should contain the address of the memory section in question. If Type is passed as zero, then DITS_SHARE_GLOBAL is assumed.

UNIX: The mapped section can be created as System V shared memory, in which case Type should be DITS_SHARE_SHMEM, or as a file accessed through mmap(), in which case Type should be DITS_SHARE_MMAP. If Type is DITS_SHARE_SHMEM, Key specifies the identifier for the shared memory and Name should be a Null string. If Type is DITS_SHARE_MMAP, Name should be the full name of the file, and Key is ignored. Address is ignored in both cases. If Type is passed as zero, DITS will assume DITS_SHARE_MMAP on systems that support mmap() and will assume DITS_SHARE_SHMEM on other systems.

C.18 DitsDeleteTask — Delete a task that is registered with the system.

Function: Delete a task that is registered with the system.

Description: This routine deletes a task from the system. The interface to this routine allows for the possibility (supported by many systems) that there is a 'polite' way to do this - one that allows the task to find out that it is being deleted, usually through some form of signal handler or exit handler - and a less polite way, which allows the task no chance to intercept the deletion but which is therefore guaranteed to work. The 'Force' parameter

can be set to indicate that the deletion of the process is to be guaranteed; if it is set false (zero) then the 'polite' form of deletion, if supported by the system, will be used. Most systems take a certain amount of time to delete a process, so even if a process is successfully deleted it may still show as present for a short time before vanishing from the system, and a calling routine must be prepared for this. If the process to be deleted is on a remote machine, then a task delete request is sent to the `IMP` 'Master' task on that remote machine. If the task was not known to the system, the Known parameter will be returned set false - but this will not cause Status to be returned indicating an error.

If as a result of a call to this routine the target task exits, then any task with outstanding actions to/from that task will be notified.

Note, this function is implemented as a macro calling ImpDeleteTask

Language: C

Call:

(Void) = DitsDeleteTask(TaskName,Force,Known,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) TaskName (char *) A null-terminated string giving the name of the target task. This should be the name supplied by that task when it called DitsInit. The case used is significant.

(>) Force (int) Set true (non-zero) if the task deletion is to be 'forced' as opposed to 'polite'.

(<) Known (int *) Set true (non-zero) to indicate that the task was known to the system. If the task is not known, this argument is set false (zero).

(!) status (StatusType *) Modified status.

Include files: DitsInteraction.h

External functions used:

ImpDeleteTask	Imp	Delete a task.

External values used: None

Prior requirements: DitsAppInit must have been called.

See Also: The Dits Specification Document, Imp Manual.

Support: Tony Farrell, `AAO`

C.19 DitsDeltaTime — Converts a time in secs and microsecs into a form usable by DitsPutDelay.

Function: Converts a time in secs and microsecs into a form usable by DitsPutDelay.

Description: The DITS routines, which have to run on a variety of machines with different internal ways of representing time values, use an internal form for a time delay that is efficient on the machine they are running on. Usually, the same value is needed many times, so it is more efficient to convert it into the internal form once rather than, for example, each time DitsPutDelay is called. So those routines that take a delta time parameter expect it passed as a structure of type DitsDeltaTime - a structure whose content is machine specific. This routine converts a time given as a number of seconds and a number of microseconds into a structure of this type. The delta time used is the total of the time in seconds and that in microseconds. The fact that a delta time can be specified to a resolution of a microsecond does not imply that this is the accuracy to which the system works - some systems can only specify time intervals in whole numbers of seconds.

Note, this function is implemented as a macro in C.

Language: C

Call:

(Void) = DitsDeltaTime (Secs,Microsecs,DeltaTime)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) Secs (unsigned int) The whole number of seconds to be used.

(>) Microsecs (unsigned int) The number of microseconds to be used. Note that this can be more than 1000000 - the two time values will be added correctly.

(<) DeltaTime (DitsDeltaTimeType *) A structure set to an efficient representation of the time specified.

Include files: DitsFix.h

External functions used:

ImpDeltaTime	Imp	Convert a time to Imp internal format

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine

See Also: The Dits Specification Document, Imp Manual, DitsPutDelay(3), GitPutDelay(3), GitPutDelayPar(3).

Support: Tony Farrell, AAO

C.20 DitsDumpAction — Dump action details.

Function: Dump action details.

Description: A debugging routine used to dump details of the specfied action.

Language: C

Call:
(Void) = DitsDumpAction (actIndex,output_routine,client_data,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) actIndex (unsigned int) The index of the action to dump.

(>) routine (DitsOutputRoutineType) The routine to invoked to perform the output. If supplied as `NULL`, then we use fprintf to stderr. Called for each line of output.

(>) client_Data (void *) Client data item fro the output routine.

(!) status (StatusType *) Modified status.

Function Prototypes Used: typedef `DVOID` (*DitsOutputRoutineType)(`DVOIDP` client_data, `DCONSTV` char *string, StatusType *status);

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) client_data (void *) The value passed to DitsDumpAction().

(>) string (const char *) The string to be output.

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used: None

External values used: DitsTask

Prior requirements: DitsAppInit should have been called.

Support: Tony Farrell, `AAO`

C.21 DitsDumpAllActions — Dump details of all actions or all active actions.

Function: Dump details of all actions or all active actions.

Description:

Language: C

Call:
(void) = DitsDumpAllActions (active, routine, client_data, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) active (int) If true, only dump active actions.

(`>`) routine (DitsOutputRoutineType) The routine to invoked to perform the output. If supplied as `NULL`, then we use fprintf to stderr. Called for each line of output.

(`>`) client_data (void *) Client data item fro the output routine.

(`!`) status (StatusType *) Modified status.

Function Prototypes Used: typedef `DVOID` (*DitsOutputRoutineType)(`DVOIDP` client_data, `DCONSTV` char *string, StatusType *status);

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) client_data (void *) The value passed to DitsDumpAllActions().

(`>`) string (const char *) The string to be output.

(`!`) status (StatusType *) Modified status.

Include files: DitsUtil.h

External functions used:

External values used: DitsTask

Prior requirements: Can only be called after DitsInit()/DitsAppInit()

Support: Tony Farrell, `AAO`

History: 07-Nov-2001 - `TJF` - Original version.

`DPUBLIC DVOIDF` DitsDumpAllActions (int active, DitsOutputRoutineType routine, `DVOIDP` client_data, StatusType *status) char buffer[200]; register int i; if (*status `!= STATUS__OK`) return;

if (!routine) routine = Dits___SimpleOutput;

First count the number of active actions to make the messages look better.

if (active) int numActive = 0; for (i = 0; i < TheTask(totalActCount) ; ++i) if (TaskActionArray[i].active) ++numActive; if (numActive == 0) ErsSPrintf(sizeof(buffer),buffer, "There are currently no active actions in task %s", TheTask(name.n)); (*routine)(client_data,buffer,status return;

Output message about dump.

ErsSPrintf(sizeof(buffer),buffer,"Dump of %sactions in task %s", (active ? "active " : ""), TheTask(name.n)); (*routine)(client_data,buffer,status);

if (TheTask(origActCount) == TheTask(totalActCount)) ErsSPrintf(sizeof(buffer),buffer, " Task supports %ld different actions.", TheTask(origActCount));

else ErsSPrintf(sizeof(buffer),buffer, " Task supports %ld different actions and has %ld entries for spawned actions", TheTask(origActCount), TheTask(totalActCount) - TheTask(origActCount));

(*routine)(client_data,buffer,status);

Dump each action we are interested in.

for (i = 0; i < TheTask(totalActCount) ; ++i) if ((active)&&(TaskActionArray[i].active)) DitsDumpAction(i, routine, client_data, status); else if (!active) DitsDumpAction(i, routine, client_data, status);

C.22 DitsDumpAllPaths — Dump all path details.

Function: Dump all path details.

Description: A debugging routine used to dump details of all known paths.

Language: C

Call:
(Void) = DitsDumpAllPaths (output_routine,client_data,status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) routine (DitsOutputRoutineType) The routine to invoked to perform the output. If supplied as `NULL`, then we use fprintf to stderr. Called for each line of output.

(>) client_Data (void *) Client data item fro the output routine.

(!) status (StatusType *) Modified status.

Function Prototypes Used: typedef `DVOID` (*DitsOutputRoutineType)(`DVOIDP` client_data, `DCONSTV` char *string, StatusType *status);

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) client_data (void *) The value passed to DitsDumpAllPaths().

(>) string (const char *) The string to be output.

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used: None

External values used: DitsTask

Prior requirements: DitsAppInit should have been called.

Support: Tony Farrell, `AAO`

C.23 DitsDumpAllTransIds — Dump all transaction id details.

Function: Dump all transaction id details.

Description: A debugging routine used to dump details of all outstanding transaction.

Language: C

Call:
(Void) = DitsDumpAllTransIds (output_routine,client_data,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) routine (DitsOutputRoutineType) The routine to invoked to perform the output. If supplied as `NULL`, then we use fprintf to stderr. Called for each line of output.

(>) client_Data (void *) Client data item fro the output routine.

(!) status (StatusType *) Modified status.

Function Prototypes Used: typedef `DVOID` (*DitsOutputRoutineType)(`DVOIDP` client_data, `DCONSTV` char *string, StatusType *status);

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) client_data (void *) The value passed to DitsDumpallTransIds().

(>) string (const char *) The string to be output.

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used: None

External values used: DitsTask

Prior requirements: DitsAppInit should have been called.

Support: Tony Farrell, `AAO`

C.24 DitsDumpImp — Starts dumping imp messages to a log file.

Function: Starts dumping imp messages to a log file.

Description: This routine arranges for all calls for Imp routines which send and receive messages, as used by `DRAMA`, to be intercepted and the message details logged to a file. This file can later be analyzed by the dumpana command.

Language: C

Call:
(void) = DitsDumpImp (char *file, StatusType *status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) file (char `*`) Name of the file to log to. If null, then a default name based on the task name is used.

(`!`) status (StatusType `*`) Modified status.

Include files: ditsdump.h

Support: Tony Farrell, `AAO`

C.25 DitsDumpImpClose — Stops dumping imp messages to a log file.

Function: Stops dumping imp messages to a log file.

Description: Closes the dump file and reverts to useing the original Imp calls for messages.

Language: C

Call:
(void) = DitsDumpImpClose (StatusType *status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`!`) status (StatusType `*`) Modified status.

Include files: ditsdump.h

Support: Tony Farrell, `AAO`

C.26 DitsDumpPath — Dump path details.

Function: Dump path details.

Description: A debugging routine used to dump details of the specfied path.

Language: C

Call:
(Void) = DitsDumpPath (path,output_routine,client_data,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) path (DitsPathType) The path to dump

(>) **routine (DitsOutputRoutineType)** The routine to invoked to perform the output. If supplied as `NULL`, then we use fprintf to stderr. Called for each line of output.

(>) **client_Data (void *)** Client data item fro the output routine.

(!) **status (StatusType *)** Modified status.

Function Prototypes Used: typedef `DVOID` (*DitsOutputRoutineType)(`DVOIDP` client_data, `DCONSTV` char *string, StatusType *status);

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) **client_data (void *)** The value passed to DitsDumpPath().

(>) **string (const char *)** The string to be output.

(!) **status (StatusType *)** Modified status.

Include files: DitsSys.h

External functions used: None

External values used: DitsTask

Prior requirements: DitsAppInit should have been called.

Support: Tony Farrell, `AAO`

C.27 DitsDumpTransId — Dump transaction id details.

Function: Dump transaction id details.

Description: A debugging routine used to dump details of the specfied transaction id.

Language: C

Call:
(Void) = DitsDumpTransId (transid,output_routine,client_data,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) **transid (DitsTransIdType)** The transaction id to dump

(>) **routine (DitsOutputRoutineType)** The routine to invoked to perform the output. If supplied as `NULL`, then we use fprintf to stderr. Called for each line of output.

(>) **client_Data (void *)** Client data item fro the output routine.

(!) **status (StatusType *)** Modified status.

Function Prototypes Used: typedef `DVOID` (*DitsOutputRoutineType)(`DVOIDP` client_data, `DCONSTV` char *string, StatusType *status);

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) client_data (void *) The value passed to DitsDumpTransId().

(>) string (const char *) The string to be output.

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used: None

External values used: DitsTask

Prior requirements: DitsAppInit should have been called.

Support: Tony Farrell, AAO

C.28 DitsEnableTask — Enable Dits calls within an interrupt handler.

Function: Enable Dits calls within an interrupt handler.

Description: Some Systems, such as VxWorks based systems, run all programs in a common address space. In such systems static and global variables can be seen by all tasks. To allow tasks to have private copies of static and global variables it is possible to have such variables saved and restored during task context switching. Dits uses this technique to store task specific information. In such systems it is sometimes necessary to call Dits routines outside the context of a task (say in an interrupt routine), during which the task specific information will be unavailable. This routine is used in conjunction with DitsGetTaskId and DitsRestoreTask to make the task specific information available in such places.

This call is normally made at the begining of an interrupt handler. The argument should be a value previously returned by a call made to DitsGetTaskId() when executing in normal task context. After this call is made, Dits routines will work as if in the task which called DitsGetTaskId(). Note that the only routine which is normally used is DitsSignalByName()/DitsSignalByIndex. Any routine which expects an action context (uface or obey/kick) should not be used since such a context may not exist. This includes Ers and MsgOut routines. Neither should DitsUfaceCtxEnable() be used, since these could clobber the current context of the main line code.

The value returned by this function should be supplied to DitsRestoreTask after before you exit the interrupt handler to ensure the task that was interrupted is restored to its original state.

This call is not neccessary on systems with process specific address spaces (VMS, UNIX, WIN32). On such systems it does nothing.

On systems such as VxWorks, it is possible to use this routine in conjunction the the DitsSignal() routine to provide communication between VxWorks tasks. For example, if one task is DRAMA and the other is not, you may use this to allow the non-DRAMA task to signal the DRAMA task. But you must ensure that A. The time between the call

to DitsEnableTask() and DitsRestoreTask() is small, say just before and after the call to DitsSignal(). B. That your `DRAMA` task will not attempt to run during this period. This could be achieved using task priorities or using the VxWorks calls taskLock() and taskUnlock() routines.

Note, this function is implemented as a macro.

Language: C

Call:
(Void) = DitsEnableTask (TaskId, SavedTaskId)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) TaskId (DitsTaskIdType) A value returned by DitsGetTaskID.

(<) SavedTaskId (DitsTaskIdType *) The value which is to be passed to DitsRestoreTask is put here.

Include files: DitsSys.h

External functions used: None

External values used: DitsTask

Prior requirements: DitsAppInit should have been called.

See Also: The Dits Specification Document, DitsGetTaskId(3), DitsRestoreTask(3), DitsSignalByName(3), DitsSignalByIndex(3).

Support: Tony Farrell, `AAO`

C.29 DitsErrorText — Converts a error string to text and returns the address of the string.

Function: Converts a error string to text and returns the address of the string.

Description: This routine is a simple interface to MessGetMsg, using an internal buffer to simplify the interface. The buffer is allocated on the first call and is overwriten on subsequent calls.

If status is ok, a simple message is written to the string.

Language: C

Call:
(char *) = DitsErrorText (status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) status (StatusType) The status too convert.

Include File: DitsUtil.h

External functions used:

MessGetMsg	Mess	Get an error message.
strcpy	CRTL	Copy one string to another.

External values used: None

Prior requirements: For an error code to be translated correctly, it must be understood by the Mess routines. DitsAppInit must have been called.

See Also: The Dits Specification Document, MessGetMsg(3).

Support: Tony Farrell, AAO

C.30 DitsFindTaskByType — Return the name of the first task with the given type.

Function: Return the name of the first task with the given type.

Description: This routine examines the types of all tasks known to the current machine. It returns the name of the first task of the given type.

Language: C

Call:
(void) = DitsFindTaskByType(type,namelen,name.found,status)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(>) type (int) The type to search for

(>) namelen (int) The number of bytes available in name.

(<) name (char *) The name is returned here. Should be at least DITS_C_NAMELEN long.

(<) found (int *) Set true if a task if found, false otherwise

(!) status (StatusType) Modified status

Include File: DitsInteraction.h

External functions used:

ImpGetDetails	Imp	Returns the details of a registered task

External values used: DitsTask - Details of the current task

Prior requirements: DitsAppInit must have been called.

See Also: The Dits Specification Document, DitsSetDetails(3), DitsScanTasks(3), DitsGetTaskDescr(3), DitsGetTaskType(3).

Support: Tony Farrell, AAO

C.31 DitsFmtReason — Converts a reason code and status into a string and outputs an

Function: Converts a reason code and status into a string and outputs an appropriate message into a buffer.

Description: This is a useful debugging routine. The reason code is converted to a string and output to a supplied string.. If the code is a failure code, then the status is also output.

Language: C

Call:
(Void) = DitsFmtReason (reason, reasonstat, bufLen, buffer, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) reason (DitsReasonType) The reason code

(>) reasonstat (StatusType) The status associate with the reason

(>) bufLen (int) The length of the output buffer.

(<) buffer (char *) The output buffer into which to put the string.

(!) status (StatusType *) Modified status.

Include files: DitsFix.h

Prior requirements: none

See Also: The Dits Specification Document, DuiLogEntry(3), DitsGetEntInfo(3), DitsPrintReason(3).

Support: Tony Farrell, `AAO`

C.32 DitsForEachTransaction — Iterate through the transactions for an action.

Function: Iterate through the transactions for an action.

Description: This fuctions allows an operation to be applied to every transaction assoicated with the current action. This is done immediately (before this function returns).

Language: C

Call:
(void) = DitsForEachTransaction (routine, client_data, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) **routine (DitsTransactRoutineType)** A routine to be invoked for each transaction. Arguments will be the client_data item below and the transaction id. Note that it IS safe to call DitsForget() on the transaction from routine.

(>) **client_data (void *)** Pass through to the routine.

(!) **status (StatusType *)** modified status, ignored.

Function Prototypes Used: typedef DVOID (*DitsTransactRoutineType) (DVOIDP client_data, DitsTransIdType tid, StatusType *status);

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

(>) **client_data (void *)** The value passed to DitsForEachTransaction().

(>) **tid (DitsTransIdType)** The transaction id.

(!) **status (StatusType *)** Modified status.

Include files: DitsUtil.h

External functions used:

External values used: DitsTask

Prior requirements: Can only be called from action routines.

See Also: The Dits Specification Document.

Support: Tony Farrell, AAO

C.33 DitsForget — Forget about the specified transaction.

Function: Forget about the specified transaction.

Description: Removes the specified transaction from the list of those transactions which will result in an entry to the current action.

If an orphan handling action exists, the transaction will be handed to it, otherwise, the transaction is made an orphan.

If the transaction already belongs to the orphan handling action, then it will not be forgotton.

Language: C

Call:
(void) = DitsForget (transid,status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

(>) **transid (DitsTransIdType)** The transaction to forget.

(!) status (StatusType *) Modified status.

Include File: DitsOrphan.h

External functions used:

Dits___TransIdRemove	Dits internal	Remove a transaction from a list

External values used: DitsTask

Prior requirements: DitsAppInit must have been called

See Also: The Dits Specification Document, DitsCheckTransactions(3), DitsIsOrphan(3), DitsTakeOrphans(3), DitsPutOrphanHandler(3).

Support: Tony Farrell, AAO

C.34 DitsGetActData — Get the value stored by DitsPutActData.

Function: Get the value stored by DitsPutActData.

Description: This call allows data to be stored between reschedules of an action and even between different invocations of the same action. This is of particular interest on machines with a common address space accross all tasks, e.g. VxWorks.

The result is zero if DitsPutActData has not been called by the current action.

Note, this function is implemented as a macro.

Language: C

Call:
(Void *) = DitsGetActData ()

Parameters: (">" input, "!" modified, "W" workspace, "<" output) none

Function value: The value stored by DitsPutActData.

Include files: DitsFix.h

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine

See Also: The Dits Specification Document, DitsPutActData(3), DitsGetUserData(3), DitsGetTransData(3), DitsGetPathData(3).

Support: Tony Farrell, AAO

C.35 DitsGetActDescr — Fetch the description of the current action.

Function: Fetch the description of the current action.

Description: Returns the description of the current action. The action is a string that can be provided when the action is created. It is meant to work as a simple help string for an action. It can be changed using DitsPutActDescr() or DitsPutThisActDescr().

When DitsFix.h.h is included by C++ module, an additional overloaded interface is provided. The overloaded version just returns a pointer to the name. In this case, if invoked when not in an action routine, a pointer to a null string is returned.

Language: C

Call:
(Void) = DitsGetActDescr (len, descr, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) len (Int) The size of descr in bytes. To ensure it gets the full length, it should be at least `DITS_ACT_DESCR_LEN`+1 bytes long (80 bytes)

(<) descr (char *) The desciption of the current action is copied here. It will be null terminated (which may cause truncation if is less then specified) the null can fit in.

(!) status (StatusType *) Modified status.

Include files: DitsFix.h Additional interface under C++ (const char *) = DitsGetActDescr ()

External functions used:

strncpy	`CRTL`	Copy one string to another.

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine

See Also: The Dits Specification Document, DitsPutActions(3), DitsPutAction(3).

Support: Tony Farrell, `AAO`

C.36 DitsGetActIndex — Return the index of the current action

Function: Return the index of the current action

Description: Returns the index of the current action. This value should only be used as an argument to DitsSignalByIndex, DitsKillByIndex or when creating an argument structure to be used to kick this action, if this action is a spawned action. In that case, the argument item should be named KickByIndex.

Language: C

Call:
(long int) = DitsGetActIndex ()

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output) none

Include files: DitsFix.h

Return value: The index of the current action.

External functions used: none

External values used: DitsTask - Details of the current task

Prior requirements: None, but if not called from an action routine, it's value is invalid.

See Also: The Dits Specification Document, DitsSignalByIndex(3), DitsKillByIndex(3), DitsSpawnKickArg(3).

Support: Tony Farrell, `AAO`

C.37 DitsGetActNameFromIndex — Given a `DITS` Action index, return the name.

Function: Given a `DITS` Action index, return the name.

Description: Given a `DITS` Action index, return the name.

Language: C

Call:
(void) = DitsGetActNameFromIndex (actIndex, len, name, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) actIndex (long) action index.

(>) len (int len) amount of space in name.

(>) name (char *) Where to write the name.

(!) status (StatusType *) Modified status.

Include File: DitsUtil.h

Prior requirements: DitsAppInit() must have been called.

See Also: The Dits Specification Document.

Support: Tony Farrell, `AAO`

C.38 DitsGetArgument — Fetch the argument supplied to this action.

Function: Fetch the argument supplied to this action.

Description: Returns the action argument structure id. The Action argument structure is encoded in Sds, hence this routine simply fetches the Sds Id of the argument. There are no restrictions on the argument, any value which can encoded in Sds may be supplied as an argument although there may be limitations to the size of messages which can be sent. See DitsAppInit(3) and DitsPathGet(3) for details of such limitations. Normally, arguments are encoded and decoded using the `ARG` package, but this is not required - other Sds routines can be used is desired.

Note that the argument structure which the returned id refers to will be deleted when this action reschedules or exits. If you want to keep it, you must make a copy (say using SdsCopy()). Also note that the id will normally refer to an External Sds item.

Note, this function is implemented as a macro.

Language: C

Call:
(SdsIdType) = DitsGetArgument ()

Include files: DitsFix.h

Function value: The argument to the action. This is an Sds `ID`. If no argument was supplied, then 0 will be returned.

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or a user interface response routine

See Also: The Dits Specification Document, DitsPutArgument(3), DitsGetSigArg(3), Sds manual.

Support: Tony Farrell, `AAO`

C.39 DitsGetCode — Get the action code of the current action.

Function: Get the action code of the current action.

Description: This function returns the code associated with the current action when DitsPutActionHandlers/ DitsPutActions was called or that was put by DitsPutCode.

Note, this function is implemented as a macro.

Language: C

Call:
(long int) = DitsGetCode ()

Include files: DitsFix.h

Function value: The action code.

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or a user interface response routine.

See Also: The Dits Specification Document, DitsPutActions(3), DitsPutActionHandlers(3), DitsPutCode(3).

Support: Tony Farrell, AAO

C.40 DitsGetContext — Get the action context of the current action.

Function: Get the action context of the current action.

Description: Fetches the action context from task common. The context indicates what type of message invoked the current reschedule of the action.

Note, this function is implemented as a macro.

Language: C

Call:
(DitsContextType) = DitsGetContext ()

Include files: DitsFix.h

Function value: The action context, one of

DITS_CTX_OBEY	Normal entry caused by either an Obey message or a normal reschedule.
DITS_CTX_KICKED	Action kicked by reception of a kick message from another task.
DITS_CTX_UFACE	User interface context, only occurs after a call to DitsUfaceCtxEnable or in a user interface response routine.

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or when the context if `DITS_CTX_UFACE`.

See Also: The Dits Specification Document, DitsPutActions(3), DitsPutActionHandlers(3).

Support: Tony Farrell, `AAO`

C.41 DitsGetDebug — Returns the value of the internal debugging flag.

Function: Returns the value of the internal debugging flag.

Description: Returns the value of the internal debugging flag, set by DitsSetDebug().

Language: C

Call:
(int) = DitsGetDebug ()

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

Include files: DitsSys.h

Return value: A mask of debugging levels. See DitsSetDebug(3) for details.

Prior requirements: DitsAppInit() should have been invoked.

Support: Tony Farrell, `AAO`

C.42 DitsGetEntBulkInfo — Allows you to retrieve bulk data transfer progress information.

Function: Allows you to retrieve bulk data transfer progress information.

Description: If the action handler has been rescheduled with a reason of `DITS_REA_BULK_TRANSFERRED` or `DITS_REA_BULK_DONE`, then the action is being notified about the progress of a bulk transfer operation.

This routine can be used to fetch details about the bulk data transfer.

The fields in BulkInfo of interest are as follows

TotalBytes	This is the total number of bytes to be transferred. Taken from the original sending call.
TransferredBytes	This is the number of bytes transferred so far.

For messages of type `DITS_REA_BULK_DONE`, these values should be equial.

Language: C

Call:
(void) = DitsGetEntBulkInfo(BulkInfo,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(<) BulkInfo (DitsBulkInfoType *) The details of the bulk data are returned here.

(!) status (StatusType *) Modified status.

Include files: DitsBulk.h

See Also: DitsInitMessBulk(3), DitsTriggerBulk(3), DitsGetEntReason(3),

Support: Tony Farrell, `AAO`

C.43 DitsGetEntComplete — Indicates if a subsidary transaction is complete.

Function: Indicates if a subsidary transaction is complete.

Description: This item may or may not be defined for the current entry, depending on the Reason for the entry. See DitsGetEntInfo for details.

If this reason for the entry relates to a transaction started by this task, this routine returns true if the transaction is complete.

Note, this function is implemented as a macro.

Language: C

Call:
(int) = DitsGetEntComplete()

Function Value: True if the transaction is complete.

Include files: DitsInteraction.h

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or a user interface response routine.

See Also: The Dits Specification Document, DitsPutActions(3), DitsPutActionHandlers(), DitsInitiateMessage(3), DitsSignalByName(3), DitsSignalByIndex(3), DitsGetEntInfo(3), DitsGetReason(3), DitsGetEntStatus(3), DitsGetEntReason(3), DitsGetEntName(3), DitsGetEntPath(3).

Support: Tony Farrell, `AAO`

C.44 DitsGetEntInfo — Get the details of the current entry of the current action.

Function: Get the details of the current entry of the current action.

Description: This routine provides all available information about the reason for entry of the current action. See `DitsGetReason` for a routine which only returns `reason` and `reasonStat`.

The `reason` argument indicates the reason for the entry and the other arguments may or may not be defined depending upon the reason code. The the `reasonStat` variable will contain the relevant completion status where appropriate.

Arguments not explicitly defined for a particular reason will be zero or null.

Note, this function is implemented as a macro.

Language: C

Call:

(Void) = DitsGetEntInfo (namelen, name, path, transid, reason, reasonStat , status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) namelen (Int) The size of name in bytes, should be at least `DITS_C_NAMELEN` bytes (20 bytes).

(<) name (char *) The name associated with the current entry. It will be null terminated, assuming the null can fit in.

(<) path (DitsPathType *) If the entry was caused by a message associated with a transaction initiated by this aciton, then this is the path to the task concerned.

(<) transid (DitsTransIdType *) If the entry was caused by a message associated with a transaction initiated by this aciton, then this is the transaction id.

(<) reason (DitsReasonType *) One of

DITS_REA_OBEY	Obey message received.
DITS_REA_KICK	Kick message received.
DITS_REA_RESCHED	Action rescheduled by timer expiry.
DITS_REA_TRIGGER	Action triggered by subsidiary action. `path` will specify the path to the task, `transid` is the transaction id return by the initiating DitsObey call and `name` will contain the name of the subsidiary action.
DITS_REA_ASTINT	Action triggered by DitsSignal. Depreciated, use DITS_REA_SIGNAL.
DITS_REA_SIGNAL	Action triggered by DitsSignal
DITS_REA_LOAD	Action triggered by a subsidiary task loading successfully. `transid` is the transaction id returned by the initiating load call and `name` is name the task has registered as.

(<) reason (DitsReasonType *) cont

DITS_REA_LOADFAILED	Action triggered by a subsidiary task failing to load. `transId` is the transaction id returned by the initiating load call `reasonStat`, `DitsLoadErrorText()` and `DitsLoadErrorStat()` can be used to get more information.
DITS_REA_MESREJECTED	A message sent to another task was rejected by that task. `path` contains the path to the task, `transid` is the transaction id returned by the initiating obey call. This is also returned if a send error occurs when processing a load request, in which case only `transid` is valid. `reasonStat` will contain additional information.
DITS_REA_COMPLETE	A `OBEY/KICK/GET/SET/CONTROL` message has completed. For `OBEY`'s, this indicateds the subsidiary action has completed. In the case of a `KICK` message with a bulk data argument, this code may be received before `DITS_REA_BULK_DONE`. `path` contains the path to the task, `transid` is the transaction id returned by the initiating obey call and `reasonStat` will contain the subsidiary action's `name` is the name of the subsidary action. completion status.
DITS_REA_DIED	A task with whom there is an outstanding transaction has died. `path` contains the path to the task, `transid` is the transaction id returned by the initiating obey call `reasonStat` will contain additional status information. `name` is the name of the subsidary action.
DITS_REA_PATHFOUND	The path to another task has been found. `path` will contain the path to the task, `transid` will contain the transaction id return by the initiating get path call and `name` will contain the name of the task.
DITS_REA_PATHFAILED	An attempt to get a path to another task failed. `path` will contain the path to the task, `transid` will contain the transaction id return by the initiating get path call and `name` will contain the name of the task. `reasonStat` will contain additional information.

(<) reason (DitsReasonType *) cont

DITS_REA_MESSAGE	A message resuling from a call to MsgOut has been received. path contains the path to the task and transid contains the transaction id returned by the initiating DitsObey call. Use DitsGetArgument to obtain the value. name is the name of the subsidary action.
DITS_REA_ERROR	A message resuling from a call to ErrFlush(or ErrOut) has been received. path contains the path to the task and transid contains the transaction id returned by the initiating DitsObey call. Use DitsGetArgument to obtain the value. name is the name of the subsidary action.
DITS_REA_EXIT	A Task loaded by this task has exited. Assuming the task was loaded successfully, this message indicates the program has now existed for whatever reason and the transaction is complete. transid contains the transaction id returned by the initiating DitsLoad call. reasonstat will be STATUS__OK if the program return a success status of DITS__EXITERR otherwise. In the later case, DitsLoadErrorText() and DitsLoadErrorStat() can be used to get more information. name is the name of the subsidary action.
DITS_REA_NOTIFY	Indicates the entry is a response to a call to DitsRequestNotify, indicating the target task's buffers are now empty. transid contains the transaction id returned by the initiating DitsRequestNotify call while path indicates the path concerned.
DITS_REA_BULK_TRANSFERRED	A notification of bulk data transfer progress. transid contains the transaction id returned by the initiaiting bulk transfer message call. Use DitsGetEntBulkInfo(3) to obtain more information. In the case of a KICK message with bulk data argument, this may be received after DITS_REA_COMPLETE.
DITS_REA_BULK_DONE	A notifcation that a bulk data transfer is complete. transid contains the transaction id returned by the initiaiting bulk transfer message call. Use DitsGetEntBulkInfo(3) to obtain more information. In the case of a KICK message with bulk data argument, this may be received after DITS_REA_COMPLETE.

(<) reasonStat (StatusType *) When the reason is a failure, this will contain the associated status of the failure.

(!) status (StatusType *) Modified status.

Include files: DitsInteraction.h

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or a user interface response routine.

See Also: The Dits Specification Document, DitsPutActions(3),a DitsPutActionHandlers(3), DitsInitiateMessage(3), DitsSignalByName(3), DitsSignalByIndex(3), DitsGetReason(3), DitsGetEntStatus(3), DitsGetEntReason(3), DitsGetEntName(3), DitsGetEntPath(3), DitsGetEntTransId(3) DitsGetEntBulkInfo(3), DitsGetEntComplete(3), DitsInitMessBulk(3).

Support: Tony Farrell, AAO

C.45 DitsGetEntName — Get the name associated with the current entry of the current action.

Function: Get the name associated with the current entry of the current action.

Description: This item may or may not be defined for the current entry, depending on the Reason for the entry. See DitsGetEntInfo for details.

Note, this function is implemented as a macro.

Language: C

Call:
(const char *) = DitsGetEntName()

Function Value: A pointer to the name assocaited with the current action. This is a null terminated string which will be valid the action until the action reschedules.

Include files: DitsInteraction.h

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or a user interface response routine.

See Also: The Dits Specification Document, DitsPutActions(3), DitsPutActionHandlers(3), DitsInitiateMessage(3), DitsSignalByName(3), DitsSignalByIndex(3), DitsGetEntInfo(3), DitsGetReason(3), DitsGetEntStatus(3), DitsGetEntReason(3), DitsGetEntPath(3), DitsGetEntTransId(3)

Support: Tony Farrell, `AAO`

C.46 DitsGetEntPath — Get the path associated with the current entry of the current action.

Function: Get the path associated with the current entry of the current action.

Description: This item may or may not be defined for the current entry, depending on the Reason for the entry. See DitsGetEntInfo for details.

Note, this function is implemented as a macro.

Language: C

Call:
(DitsPathType) = DitsGetEntPath()

Function Value: The path.

Include files: DitsInteraction.h

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or a user interface response routine.

See Also: The Dits Specification Document, DitsPutActions(3), DitsPutActionHandlers(3), DitsInitiateMessage(3), DitsSignalByName(3), DitsSignalByIndex(3), DitsGetEntInfo(3), DitsGetReason(3), DitsGetEntStatus(3), DitsGetEntReason(3), DitsGetEntName(3), DitsGetEntTransId(3)

Support: Tony Farrell, `AAO`

C.47 DitsGetEntReason — Get the reason for the current entry of the current action.

Function: Get the reason for the current entry of the current action.

Description: Fetch the reason for this entry from task common. This routine provides an alternative interface to information available from DitsGetReason and DitsGetEntInfo.

Note, this function is implemented as a macro.

Language: C

Call:
(DitsReasonType) = DitsGetEntReason ()

Function value: One of-

DITS_REA_OBEY	Obey message received
DITS_REA_KICK	Kick message received
DITS_REA_RESCHED	Action reschedule by timer expiry
DITS_REA_TRIGGER	Action triggered by subsidiary action
DITS_REA_ASTINT	Action triggered by signal handler Depreciated, use DITS_REA_SIGNAL.
DITS_REA_SIGNAL	Action triggered by signal handler.
DITS_REA_LOAD	Action triggered by successful load
DITS_REA_LOADFAILED	Action triggered by load failure
DITS_REA_MESREJECTED	A message was rejected
DITS_REA_COMPLETE	Message completed
DITS_REA_DIED	Task died
DITS_REA_PATHFOUND	Path to a task was found
DITS_REA_PATHFAILED	Failed to get a path
DITS_REA_MESSAGE	DITS_MSG_MESSAGE message received
DITS_REA_ERROR	DITS_ERR_MESSAGE message received
DITS_REA_EXIT	A loaded task has exited.
DITS_REA_NOTIFY	A notification request, requested using DitsRequestNotify, returned.
DITS_REA_BULK_TRANSFERRED	A notification of bulk data transfer progress.
DITS_REA_BULK_DONE	A notifcation that a bulk data transfer is complete.
DITS_REA_DRAMA2	A signal triggered by DitsSignalDrama2(), used by the DRAMA V2 implementation.

Include files: DitsFix.h

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or a user interface response routine.

See Also: The Dits Specification Document, DitsPutActions(3), DitsPutActionHandlers(3), DitsInitiateMessage(3), DitsSignalByName(3), DitsSignalByIndex(3), DitsGetEntInfo(3), DitsGetReason(3), DitsGetEntStatus(3), DitsGetEntName(3), DitsGetEntBulkInfo(3), DitsGetEntPath(3), DitsGetEntTransId(3)

Support: Tony Farrell, AAO

C.48 DitsGetEntStatus — Get the status associated with the reason for the current entry

Function: Get the status associated with the reason for the current entry of the current action.

Description: Fetch the status associated with the reason for this entry from task common. This routine provides an alternative interface to information available from DitsGetReason and DitsGetEntInfo.

Note, this function is implemented as a macro.

Language: C

Call:

(StatusType) = DitsGetEntStatus ()

Function value: The status.

Include files: DitsFix.h

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or a user interface response routine.

See Also: The Dits Specification Document, DitsPutActions(3), DitsPutActionHandlers(3), DitsInitiateMessage(3), DitsSignalByName(3), DitsSignalByIndex(3), DitsGetEntInfo(3), DitsGetReason(3), DitsGetEntReason(3), DitsGetEntName(3), DitsGetEntPath(3), DitsGetEntTransId(3), DitsGetEntBulkInfo(3).

Support: Tony Farrell, AAO

C.49 DitsGetEntTransId — Get the transaction id associated with the current

Function: Get the transaction id associated with the current entry of the current action.

Description: This item may or may not be defined for the current entry, depending on the Reason for the entry. See DitsGetEntInfo for details.

Note, this function is implemented as a macro.

Language: C

Call:
(DitsTransIdType) = DitsGetEntTransId()

Function Value: The transaction id

Include files: DitsInteraction.h

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or a user interface response routine.

See Also: The Dits Specification Document, DitsPutActions(3), DitsPutActionHandlers(), DitsInitiateMessage(3), DitsSignalByName(3), DitsSignalByIndex(3), DitsGetEntInfo(3), DitsGetReason(3), DitsGetEntStatus(3), DitsGetEntReason(3), DitsGetEntName(3), DitsGetEntPath(3).

Support: Tony Farrell, `AAO`

C.50 DitsGetFixFlags — Gets the value of the flags used to commuicate with the fixed part.

Function: Gets the value of the flags used to commuicate with the fixed part.

Description: This function gets the current values of the flags by DitsSetFixFlags()

Language: C

Call:
(int) = DitsGetFixFlags()

`DITS_M_NO_SDS_CHECK`	Don't do `SDS` leak check for this action, even if log flag is set - allows for actions which do allocate or release `SDS` ids on purpose and for lower level checking

Include files: DitsSys.h

Prior requirements: DitsAppInit() should have been invoked, must be in an action/uface context.

Support: Tony Farrell, AAO

C.51 DitsGetMsgLength — Return the buffer length required to send a given message.

Function: Return the buffer length required to send a given message.

Description: This routine will determine the size of a message containing the given argument. If reply is false, then this is considered to be a message as originted by DitsInitiateMessage() and the routines based on it (such as DitsObey() etc.)

If reply is true, then this is considered to be a message to be sent to the originator of an obey, say using DitsTrigger().

Language: C

Call:
(void) = DitsGetMsgLength(reply,argument,size,status)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(>) reply (int) If true, then check the length as a reply. (e.g. an argument to a trigger message).

(>) argument (SdsIdType) The argument to the message

(<) size (unsigned long *) The size in bytes required for the message is written here.

(!) status (StatusType) Modified status

Include File: DitsInteraction.h

External functions used:

SdsSize	Sds	Return the size of an Sds item
SdsPut	Sds	Put the value of an sds item.

External values used: DitsTask

Prior requirements: DitsAppInit must have been called

See Also: The Dits Specification Document, DitsPathGet(3), DitsAppInit(3), DitsGetPathSize(3), DitsRequestNotify(3), Sds manual.

Support: Tony Farrell, AAO

C.52 DitsGetMsgTypeStr — Return a string description of a message type code.

Function: Return a string description of a message type code.

Description: Given a `DRAMA` message type string, return a pointer to a string version of the code.

Language: C

Call:
(const char *) = DitsGetMsgTypeStr (type)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) type (DitsMsgType) The `DITS` Message type code.

Include files: DitsSys.h

External functions used: None

External values used: None

Prior requirements: None

Support: Tony Farrell, `AAO`

C.53 DitsGetName — Fetch the name of the current action.

Function: Fetch the name of the current action.

Description: Returns the name of the current action.

Note, this function is implemented as a macro.

When DitsFix.h.h is included by C++ module, this function is implemented as an inline function and an additional overloaded interface is provided. The overloaded version just returns a pointer to the name. In this case, if invoked when not in an action routine, a pointer to a null string is returned.

Language: C

Call:
(Void) = DitsGetName (namelen, name, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) namelen (Int) The size of name in bytes, should be at least `DITS_C_NAMELEN` bytes (20 bytes).

(<) **name (char *)** The name of the current action is copied here. It will be null terminated, but will be truncated if namelen is not long enough.

(!) **status (StatusType *)** Modified status.

Include files: DitsFix.h Additional interface under C++ (const char *) = DitsGetName ()

External functions used:

strncpy	CRTL	(when not under VMS) Copy one string to another. Under VMS, other techniques are used.

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine

See Also: The Dits Specification Document, DitsPutActions(3), DitsPutActionHandlers(3).

Support: Tony Farrell, AAO

C.54 DitsGetParId — Get the parameter system id supplied in the parid argument

Function: Get the parameter system id supplied in the parid argument to DitsAppParSys.

Description: Simply return the value. This function is implemented as a macro.

Language: C

Call:
(void *) = DitsGetParId()

Include files: DitsParam.h

Function value: The parameter system id.

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or when the context if DITS_CTX_UFACE.

See Also: The Dits Specification Document, DitsAppParamSys(3), SdpInit(3), SdpCreate(3).

Support: Tony Farrell, AAO

C.55 DitsGetParam — Send a "Get Parameter" message to a task

Function: Send a "Get Parameter" message to a task

Description: A `GET` message with the specified parameter name is sent to the task on the given path.

This function transparently handles long parameter names (length `> DITS_C_NAMELEN`) by converting to a `MGET` message. Should should not effect the operation of the call other then DitsGetName will return a "`_LONG_`" string instead of the parameter name when the reschedule event occurs.

If the message fails, a "transaction failure" message will be received by the task with the transaction id returned by this routine. otherwise, the current action will be rescheduled when the value is returned, hence the calling action should reschedule to await this. When the reschedule occurs, the value of the parameter can be fetched using DitsGetArgument. The format of the argument structure is dependent on the parameter system being used by the target task.

Language: C

Call:
(Void) = DitsGetParam(path,param,transid, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) path (DitsPathType) The path to the task to which the message is directed. See DitsGetPath for details on how to get a path to another task.

(>) param (char *) The null terminated name of the parameter who's value is to be returned.

(<) transid (DitsTransId *) Transaction id for the transaction started. If an address of zero is supplied, then don't create a transaction id. In this case we will never be notified of transaction errors or the value, which would seem to be a silly thing to do.

(!) status (StatusType *) Modified status.

Include files: DitsParam.h

External functions used:

DitsInitiateMessage	Dits internal	Send a message to another task.

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or when the context is `DITS_CTX_UFACE`.

See Also: The Dits Specification Document, DitsInitateMessage(3), DitsSetParam(3), DitsAppParamSys(3), DuiExecuteCmd(3), DulMessageW(3).

Support: Tony Farrell, `AAO`

C.56 DitsGetParentPath — Return the path to the task which invoked this action.

Function: Return the path to the task which invoked this action.

Description: This function returns the path to the task which invoked this action. Note that such a path is not always defined. It will not be definied if invoked when context is `DITS_CTX_UFACE` or if context is `DITS_CTX_KICKED` and DitsGetReason returns `DITS_REA_DIED`. In both these cases 0 will be returned.

Note, this function is implemented as a macro.

Language: C

Call:
(DitsPathType) = DitsGetParentPath()

Function Value: The path.

Include files: DitsInteraction.h

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or a user interface response routine.

See Also: The Dits Specification Document, DitsPathGet(3), DitsTaskFromPath(3).

Support: Tony Farrell, `AAO`

C.57 DitsGetPath — Initiates the getting of or returns a path to a task, Obsolete,

Function: Initiates the getting of or returns a path to a task, Obsolete, See DitsPathGet.

Description: A path is required for a Dits task to send messages to other Dits tasks. This routine will either return a path to the specified task or initiate actions to obtain the path.

If the task is local or known locally, then the path can normally be returned immediately. In this case, transid is set to 0.

If the task not known then some message system interaction may be required to locate the task. The path returned is the correct path to the task but it will not be valid until an appropriate message is received. The user action calling this routine should reschedule to await this message, which will have a reason code of `DITS_REA_PATHFOUND` or `DITS_REA_PATHFAILED` with the transaction id as returned by this routine.

This routine is now implemented in terms of DitsPathGet. See that routine for more details. A flag to DitsAppInit determines if flow control is to be enabled for the resulting connection.

Please note that there is only ever one path between any two tasks and the buffer sizes are as set up for the first path to be set up between the two tasks. Subsequent calls to set up a path just returns the existing path, using the original buffer sizes. We consider this a flaw and believe the buffer sizes should be the maximum specified in any such calls, but we don't have a way of implementing this at this time. Note that if a given task has been sending connections to your task, then a connection has already been set up.

Language: C

Depreciated: See DitsPathGet(3).

Call:

(Void) = DitsGetPath (name, messageBytes, maxMessages, replyBytes, maxReplies ,path, transid, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) name (char *) The null terminated name the task is thought to be registered by. The format is one of -

taskname	The task is on the local machine or is remote but already known locally.
taskname@nodename	The task is on the specified node. nodename is an internet address in either dot or domain format.

(>) messageBytes (int) The maximum number of bytes anticipated for any message to be sent.

(>) maxMessages (int) The maximum number of messages of size MaxBytes to be queued.

(>) replyBytes (int) This is the maximum number of bytes anticipated in any messages sent back to the calling task.

(>) maxReplies (int) The maximum number of replies of size ReplyBytes anticipated.

(<) path (DitsPathType *) The path id is written here. Full details of the path may or may not be known immediately. If the task to which a path is required is already known then this path will be valid immediately and transid will be 0. If The task is remote and not yet known then messages must be sent to find it. When this happens, transid will contain a transaction id and the path will not become valid until an appropriate message is received. The user's action routine should reschedule to await invocation with a reason of `DITS_REA_PATHFOUND` or `DITS_REA_PATHFAILED`.

Once created, a path is not deleted until the the current task exits, although it will become invalid if the target task disconnects.

(<) transid (DitsTransIdType *) If the path is known immediately, then this will be set to zero. If not, it will contain the transaction id of the transaction required to find the path. If not this pointer is not supplied, then not message system interaction will be started, we return an error unless the path is known immediatly.

(!) status (StatusType *) Modified status.

External functions used:

DitsPathGet	Dits	Underlying routine.

Include files: DitsInteraction.h

External values used: None

Prior requirements: Can only be called from a Dits application routine or when the context if `DITS_CTX_UFACE`.

See Also: The Dits Specification Document, DitsPathGet(3), DitsAppInit(3), DitsInititateMessage(3), DulGetPathW(3), DuiExecuteCmd(3), DitsPathGetInit(3).

Support: Tony Farrell, `AAO`

C.58 DitsGetPathData — Retrun an item previously associated with a path.

Function: Retrun an item previously associated with a path.

Description: This call returns a item associated with the specified path by a previous call to DitsPutPathData(). If DitsPutPathData() was not invoked for this path then zero will be returned.

Note, this function is implemented as a macro.

Language: C

Call:
(Void) = DitsGetPathData(path,data,status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

(>) path (DitsPathType) The path as returned by another Dits call.

(<) data (void **) The data item to associated with the path

(!) status (StatusType *) Modified status.

Include files: DitsInteraction.h

External functions used: None

External values used: None

Prior requirements: None.

See Also: The Dits Specification Document, DitsPutPathData(3), DitsGetUserData(3), DitsGetTransData(3), DitsGetPathData(3), DitsPutActData(3).

Support: Tony Farrell, AAO

C.59 DitsGetPathSize — Return message buffer allocation for the specified path.

Function: Return message buffer allocation for the specified path.

Description: For the specified path, return buffer allocation - the total number of bytes available to send messages.

Language: C

Call:
(void) = DitsGetPathSize(path,size,status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) path (DitsPathType) The path

(<) size (unsigned long *) The number of bytes available to send messages.

(!) status (StatusType) Modified status

Include File: DitsInteraction.h

External functions used: None

External values used: None

Prior requirements: DitsAppInit must have been called.

See Also: The Dits Specification Document, DitsPathGet(3), DitsAppInit(3), DitsGetMsgLength(3), DitsRequestNotify(3), Sds manual.

Support: Tony Farrell, AAO

C.60 DitsGetReason — Get the reason for the current entry of the current action, Obsolete, see DitsGetEntReason(3) and DitsGetEntStatus(3).

Function: Get the reason for the current entry of the current action, Obsolete, see DitsGetEntReason(3) and DitsGetEntStatus(3).

Description: Fetch the reason for this entry from task common.

Whilst this routine will remain available, it is probably better to use the more modern routines DitsGetEntReason(3) and DitsGetEntStatus(3).

Note, this function is implemented as a macro in C.

Language: C

Call:

(Void) = DitsGetReason (reason, reasonStat , status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(<) reason (DitsReasonType *) One of

`DITS_REA_OBEY`	Obey message received
`DITS_REA_KICK`	Kick message received
`DITS_REA_RESCHED`	Action reschedule by timer expiry
`DITS_REA_TRIGGER`	Action triggered by subsidiary action
`DITS_REA_ASTINT`	Action triggered by DitsSignal. Depreciated, use `DITS_REA_SIGNAL`.
`DITS_REA_SIGNAL`	Action triggered by DitsSignal().
`DITS_REA_LOAD`	Action triggered by successful load
`DITS_REA_LOADFAILED`	Action triggered by load failure
`DITS_REA_MESREJECTED`	A message was rejected
`DITS_REA_COMPLETE`	Message completed
`DITS_REA_DIED`	Task died
`DITS_REA_PATHFOUND`	Path to a task was found
`DITS_REA_PATHFAILED`	Failed to get a path
`DITS_REA_MESSAGE`	`DITS_MSG_MESSAGE` message received
`DITS_REA_ERROR`	`DITS_ERR_MESSAGE` message received
`DITS_REA_EXIT`	A loaded task has exited.
`DITS_REA_NOTIFY`	A notification request, requested using DitsRequestNotify, returned.
`DITS_REA_BULK_TRANSFERRED`	A notification of bulk data transfer progress.
`DITS_REA_BULK_DONE`	A notifcation that a bulk data transfer is complete.
`DITS_REA_DRAMA2`	A signal triggered by DitsSignalDrama2(), used by the `DRAMA V2` implementation.

(<) reasonStat (StatusType *) When the reason is a failure, this will contain the associated status of the failure.

(!) status (StatusType *) Modified status.

Include files: DitsFix.h

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or a user interface response routine.

See Also: The Dits Specification Document, DitsPutActions(3), DitsPutActionHandlers(3), DitsInitiateMessage(3), DitsSignalByName(3), DitsSignalByIndex(3), DitsGetEntInfo(3), DitsGetReason(3), DitsGetEntStatus(3), DitsGetEntName(3), DitsGetEntPath(3), DitsGetEntTransId(3)

Support: Tony Farrell, AAO

C.61 DitsGetSelfPath — Return the path to this task.

Function: Return the path to this task.

Description: Each task always maintains a path to itself, which can be used to send messages to the task itself - which is often usefull.

This function returns this path. The buffer size for thisi path is set during the call to DitsAppInit(3) can can be modified by the arguments to that call.

Previously (prior to DITS Version 3.25) this was often obtained by using DitsGetTaskName(3) to fetch the name of the task and then using DitsGetPath(3) to fetch the patch. This function is much simplier and more efficent.

Language: C

Call:
(DitsPathType) = DitsGetSelfPath()

Function Value: The path to the task itself.

Include files: DitsInteraction.h

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: DitsAppInit must have been called.

See Also: The Dits Specification Document, DitsGetPath(3), DitsPathGet(3), DitsAppInit(3).

Support: Tony Farrell, AAO

C.62 DitsGetSeq — Get the sequence count of the current action.

Function: Get the sequence count of the current action.

Description: Fetch the sequence count from task common. The sequence count will be set to 0 when the message initiating the current action is received. It is incremented by one for each subsequent reschedule of the action.

Note, this function is implemented as a macro.

Language: C

Call:
(Unsigned long Int) = DitsGetSeq ()

Include files: DitsFix.h

Function value: The action sequence number.

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine

See Also: The Dits Specification Document, DitsPutActions(3), DitsPutActionHandlers(3).

Support: Tony Farrell, AAO

C.63 DitsGetSigArg — Fetch the argument supplied to this action by a DitsSignal...Ptr call.

Function: Fetch the argument supplied to this action by a DitsSignal...Ptr call.

Description: The DitsSignalByNamePtr() and DitsSignalByIndexPtr() allow you to supply a pointer as one of the arguments. This function will return that pointer value or 0 if this entry was not triggered by such a signal call.

Note, this function is implemented as a macro.

Language: C

Call:
(void *) = DitsGetSigArg ()

Include files: DitsFix.h

Function value: The signal argument to the action. If no signal argument was supplied, then 0 will be returned.

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or a user interface response routine

See Also: The Dits Specification Document, DitsGetArgument(3), DitsSignalByNamePtr(3), DitsSignalByIndexPtr(3).

Support: Tony Farrell, `AAO`

C.64 DitsGetSymbol — Returns the value of an external symbol

Function: Returns the value of an external symbol

Description: Most systems provide a means of setting a named symbol to an equivalent string. `VMS` has logical names, `UNIX` and VxWorks have environment variables. These can be used to exercise control over the detailed operation of programs without needing a dialogue with the user or a change to the program itself. This routine is passed the name of such a symbol and returns the string - if any - to which it has been equivalenced.

This routine takes the name of the symbol, the address of the string into which the string to which it has been equivalenced is to be written, and the number of bytes available at that address. If it can get a value for the specified symbol, it returns that value, truncated if necessary in the string whose address it has been passed. The string it returns will always be nul-terminated. The routine returns a true (non-zero) value if it was able to translate the string (even if it was truncated), and returns a false (zero) value if it could not translate the string. On systems where no such mechanism is provided, it will always return false. The case of the symbol will be treated in the natural way for the system - it will be respected under `UNIX`, for example, but ignored under `VMS`.

Under `WIN32`, we look at environment variables first, and if a value is not found, the Environment component of the Registry examined.

Note that this function is currently implemented as a macro which invokes ImpZGetSymbol

Language: C

Call:
(int) = DitsGetSymbol (symbol, string, length)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) symbol (const char *) The symbol name

(<) string (char *) The translation is written here.

(>) length (int) Output string length

Include files: DitsSys.h

Return value: True if we were able to translate the string. False otherwise.

External functions used: ImpZGetSymbol

Prior requirements: none

Support: Tony Farrell, AAO

C.65 DitsGetTaskDescr — Return the description of a task of a given name.

Function: Return the description of a task of a given name.

Description: This routine returns the description a task set with the DitsSetDetails() call.

Note, this function is implemented as a macro.

Language: C

Call:

(Void) = DitsGetTaskDescr(name,desclen,descr,status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) name (char *) The name of the task.

(>) desclen (int) The number of bytes in desc.

(<) descr (char *) The description of the task.

(!) status (StatusType *) Modified status. DITS__UNKNTASK if the task is not known to the system. Imp error are also possible.

Include files: DitsInteraction.h

External functions used:

ImpGetDetails	Imp	Get task details.

External values used: None

Prior requirements: DitsAppInit must have been called.

See Also: The Dits Specification Document, DitsSetDetails(3), DitsScanTasks(3), DitsFindTaskByType(3), DitsGetTaskType(3).

Support: Tony Farrell, AAO

C.66 DitsGetTaskId — Get the current Dits Task Id

Function: Get the current Dits Task Id

Description: Some Systems, such as VxWorks based systems, run all programs in a common address space. In such systems static and global variables can be seen by all tasks. To allow tasks to have private copies of static and global variables it is possible to have such variables saved and restored during task context switching. Dits uses this technique to store task specific information. In such systems it is sometimes necessary to call Dits routines outside the context of a task (say in an interrupt routine), during which the task specific information will be unavailable. This routine is used in conjunction with DitsEnableTask and DitsRestoreTask to make the task specific information available in such places.

This routine is called from the task itself, not the interrupt handler. DitsGetTaskId returns the task Id of the current Dits task. This value is supplied to DitsEnableTask to make the context of this task available.

Note, this function is implemented as a macro.

Language: C

Call:
(DitsTaskIdType) = DitsGetTaskId ()

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output) None

Include files: DitsSys.h

Function Value: The Id of the current Dits program. This is the same size as (void `*`).

External functions used: None

External values used: DitsTask

Prior requirements: DitsAppInit(3) should have been called.

See Also: The Dits Specification Document, DitsEnableTask(3), DitsRestoreTask(3), DitsSignalByName(3), DitsSignalByIndex(3).

Support: Tony Farrell, `AAO`

C.67 DitsGetTaskName — Fetch the name of the task.

Function: Fetch the name of the task.

Description: Returns the name of the task.

Note, this function is implemented as a macro.

When DitsFix.h.h is included by C++ module, this function is implemented as an inline function and an additional overloaded interface is provided. The overloaded version just returns a pointer to the name.

Language: C

Call:
(Void) = DitsGetTaskName (namelen, name, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) namelen (Int) The size of name in bytes, should be at least `DITS_C_NAMELEN` bytes (20 bytes).

(`<`) name (char `*`) The name of the task is copied here. It will be null terminated, assuming the null can fit in.

(`!`) status (StatusType `*`) Modified status.

Include files: DitsFix.h Additional interface under C++ (const char `*`) = DitsGetTaskName ()

External functions used:

strncpy	`CRTL`	(when not under `VMS`) Copy one string to another. Under `VMS`, other techniques are used.

External values used: DitsTask - Details of the current task

Prior requirements: DitsAppInit must have been called.

See Also: The Dits Specification Document, DitsAppInit(3), DitsAppInit(3).

Support: Tony Farrell, `AAO`

C.68 DitsGetTaskType — Return the type of a task of a given name.

Function: Return the type of a task of a given name.

Description: This routine returns the type a task set with the DitsSetDetails() call.

Note, this function is implemented as a macro.

Language: C

Call:
(Void) = DitsGetTaskType(name,type,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) name (char *) The name of the task.

(`<`) type (int *) The type of the task.

(`!`) status (StatusType *) Modified status. `DITS_UNKNTASK` if the task is not known to the system. Imp error are also possible.

Include files: DitsInteraction.h

External functions used:

ImpGetDetails	Imp	Get task details.

External values used: None

Prior requirements: DitsAppInit must have been called.

See Also: The Dits Specification Document, DitsSetDetails(3), DitsScanTasks(3), DitsGetTaskDescr(3), DitsFindTaskByType(3).

Support: Tony Farrell, `AAO`

C.69 DitsGetTransData — Retrun an item previously associated with a transaction.

Function: Retrun an item previously associated with a transaction.

Description: This call returns a item associated with the specified transaction by a previous call to DitsPutTransData(). If DitsPutTransData() was not invoked for this transaction then zero will be returned, unless the transaction was started in uface context, in which case the uface code current at that time will be returned.

Note, this function is implemented as a macro.

Language: C

Call:

(Void) = DitsGetTransData(transid,data,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) transid (DitsTransIdType) The transaction as returned by another Dits call.

(`<`) data (void **) The data item to associated with the transaction

(`!`) status (StatusType *) Modified status.

Include files: DitsInteraction.h

External functions used: None

External values used: None

Prior requirements: None.

See Also: The Dits Specification Document, DitsPutTransData(3), DitsGetUserData(3), DitsGetPathData(3), DitsGetActData(3).

Support: Tony Farrell, `AAO`

C.70 DitsGetUserData — Get the user data put by the DitsPutUserData routines.

Function: Get the user data put by the DitsPutUserData routines.

Description: Fetches the data put by DitsPutUserData from task common.

See DitsPutUserData for more details.

Note, this function is implemented as a macro.

Language: C

Call:
(void *) = DitsGetUserData ()

Include files: DitsFix.h

Function value: The value supplied by the DitsPutUserData routines.

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Should only be called after DitsAppInit has been called.

See Also: The Dits Specification Document, DitsPutUserData(3), DitsGetTransData(3), DitsGetPathData(3), DitsGetActData(3).

Support: Tony Farrell, `AAO`

C.71 DitsGetVerDate — Returns the `DITS` version date

Function: Returns the `DITS` version date

Description: Returns a pointer to the `DITS` version date string. This string is generated fromt he C `__DATE__` marco when the function was compliled. The compilation of this module is normally linked to the `DITS` version.

Language: C

Call:
(const char *) = DitsGetVerDate ()

Include File: DitsUtil.h

External functions used: None

External values used: None

Prior requirements: None

See Also: The Dits Specification Document.

Support: Tony Farrell, AAO

C.72 DitsGetVersion — Returns the DITS version.

Function: Returns the DITS version.

Description: Returns a pointer to the DITS version string. This string is generated from the DITS version number when DITS was built.

Language: C

Call:
(const char *) = DitsGetVersion ()

Include File: DitsUtil.h

External functions used: None

External values used: None

Prior requirements: None

See Also: The Dits Specification Document.

Support: Tony Farrell, AAO

C.73 DitsGetXInfo — Returns information needed by X-windows to coexist with IMP.

Function: Returns information needed by X-windows to coexist with IMP.

Description: If an X-compatible notification mechanism is being used, that is if DitsAppInit() was called with the DITS_M_X_COMPATIBLE flag bit set, then Dits routines can be used in a program that has been written to handle X-windows events. The X system provides a routine XtAppAddInput() which allows it to be informed of other event sources. This routine returns the information that has to be passed to XtAppAddInput() in order for it to include the arrival of Dits messages as events to be handled.

The routine XtAppAddInput() requires two parameters: 'source' which is declared as an int and 'condition' which is declared as an XtPointer. The present routine is passed two corresponding arguments, the first declared as the address of an int, the second declared as the address of a pointer to void. The entities located at these addresses are set by this routine to values suitable to be passed to XtAppAddInput(). If the Dits routines for this process are not running in an X-compatible way, an error code is returned.

Note, this function is implemented as a macro.

Language: C

Call:

(Void) = DitsGetXInfo (Source,Condition,Status)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(<) Source (int *) Receives the value to be used as the source parameter in a call to XtAppAddInput().

(<) Condition (void **) Receives the value to be used as the condition parameter in a call to XtAppAddInput().

(!) Status (StatusType *) Inherited status value. If a non-zero value is passed, this task returns immediately. Otherwise, if an error is detected, an IMP__ code will be returned.

Include files: DitsSys.h

External functions used:

ImpGetXInfo	Imp	Get X-windows information.

External values used: DitsTask

Prior requirements: DitsAppInit(3) should have been called.

See Also: The Dits Specification Document, DitsAppInit(3), DitsAltInLoop(3), DtclAppMainLoop(3), DtclTkAppMainLoop(3), Imp manual.

Support: Tony Farrell, AAO

C.74 DitsImpRedirect — Redirects the ImpNetLocate and ImpConnect calls

Function: Redirects the ImpNetLocate and ImpConnect calls

Description: This routine arranges for all calls for ImpNetLocate and ImpConnect to be redirected to user routines that perform the same function. This allows the user to handle some functionality that Imp doesn't handle particularly well - for example, setting path sizes and the nodes on which machines are found.

The user must, of course, call ImpNetLocate and ImpConnect within their routines in order to do the normal Imp processing.

Users should note that DitsDumpImp also redirects the ImpNetLocate and ImpConnect calls using the same mechanism, and so the use of DitsDumpImp and DitsImpRedirect at the same time is incompatible.

To switch back to the normal Imp behaviour, call DitsImpRedirect(ImpNetLocate, ImpConnect, &status);

A NULL or 0 function pointer for one of the routines will not do anything.

Language: C

Call:
(void) = DitsImpRedirect (NetLocate, Connect, &status);

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) NetLocate (Dits___NetLocateType) A pointer to a function that is a replacement for ImpNetLocate.

(>) Connect (Dits___ConnectType) A pointer to a function that is a replacement for ImpConnect.

(!) status (StatusType *) Modified status.

Include files: DitsImpRedirect.h

Support: Tony Farrell, AAO

C.75 DitsInit — Initialise Dits, Depreciated, See DitsAppInit(3).

Function: Initialise Dits, Depreciated, See DitsAppInit(3).

Description: Calls DitsAppInit(3). This routine is provided for compatiblty with old versions. New code should use DitsAppInit(3).

See DitsAppInit(3) for more details, but beware when calling DitsInit(3) routine that you cannot set any flag which uses the initInfo structure.

Note, this function is implemented as a macro calling DitsAppInit(3).

Language: C

Call:
(Void) = DitsInit (taskname, bytes, flags, status)

Depreciated: See DitsAppInit(3).

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

(>) taskname (Char *) The name the task is to be known by.

(>) bytes (Int) This argument indicates how many bytes should be allocated to the message buffer. See

() DitsAppInit () for details.

(>) flags (Int) A flag word containing some options. See DitsAppInit().

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used:

DitsAppInit	Dits	Actual initial routine.

External values used: See DitsAppInit(3).

Prior requirements: Should be called before any other Dits calls.

See Also: The Dits Specification Document, DitsAppInit(3), DitsPutActions(3), Imp manual, Ers manual, Sds manual, Messgen manual.

Support: Tony Farrell, AAO

C.76 DitsInitMessBulk — Send a message to another task, sending a bulk data argument.

Function: Send a message to another task, sending a bulk data argument.

Description: This is a bulk data version of the general message sending routine DitsInitiateMessage(3). You can use this routine to send OBEY, KICK etc. messages.

In this version of DitsInitiateMessage(3), you can attach an amount of bulk data defined by an DITS shared memory segment. The bulk data is passed as efficiently as possible to the target task.

If the bulk data contains an exported SDS item, then the target task can receive the bulk data without doing any more work. Alternatively, and required if SDS is not involved, the target task can use various DITS inquiry functions to handle it specially.

In addition to the normal messages associated with a message transaction, you will also receive notifications about the bulk data transfer. The sending action may receive additional reschedule events with an entry reason of `DITS_REA_BULK_TRANSFERRED` indicating the progress of the transfer. It will also receive an entry with a reason of `DITS_REA_BULK_DONE` when the target task releases the shared memory. You can then call DitsGetEntBulkInfo() to retrieve details of the transfer.

The reception of `DITS_REA_BULK_TRANSFERRED` messages are dependent on the behavour of the target task. If it is a local task, then these mesages are recevid each time it calls DitsBulkArgReport(). If the target task is remote, these messages are sent by the local `IMP` transmitter task at a rate determined by it. If NotifyBytes is non-zero it indicates a hint to the target target about the rate at which these nofications are requested.

You should not delete the shared memory yourself until get the first message using one of these codes. When you get `DITS_REA_BULK_DONE` you are free to reuse all the shared memory as the target task has finished accessing the shared memory. You get reuse parts of the shared memory earily by making use of information retreived by DitsGetEntBulkInfo() to determine what parts of the bulk data have been used.

If the target task is local, then `DITS_REA_BULK_DONE` is received when the target task action entry returns or alternatively if the task invoked DitsBulkArgInfo(3), when it has called DitsBulkArgRelease(3).

If the target task is remote, then `DITS_REA_BULK_DONE` is recevied when the local transmitter task has finished forwarding the data to the remote machine.

As a result, in the remote task case, you are not notified when the target task has finished accessing the data, just when it is safe for you to re-use the local shared memory.

Full details about the parameters of this call common to DitsInitiateMessage(3) can be obtained from the documentation for that call. The only new argments are SharedMemInfo and NotifyBytes. There is an additional flag, `DITS_M_SDS`.

If the target task did not accept the message, you will receive an reschedule event with a reason of `DITS_REA_MESREJECTED` and the status returned by DitsGetEntStatus(3) will return the status associated with the rejection. This may occur before or after an reschedule event with reason of `DITS_REA_BULK_DONE`. If it occurs before, you will not receive the `DITS_REA_BULK_DONE` reschedule event.

If and only if you are sending a `KICK` message, the bulk data messages (`DITS_REA_BULK_DONE` and `DITS_REA_BULK_TRANSFFERED`) may be received after you have recived `DITS_REA_COMPLETE`. This would occur because the target action's kick handler has invoked DitsBulkArgInfo(3) but the corresponding DitsBulkArgRelease(3) occurs in the action's obey handler. As a result, the kick message completion occurs before the bulk data is released. Use the routine DitsGetEntCompleted(3) to determine if a `DITS_REA_BULK_DONE` or `DITS_REA_COMPLETE` message indicates completion. This can only happen if the target task is a local task.

Language: C

Call:

(void) = DitsInitMessBulk(flags,path,SharedMemInfo,NotifyBytes, transid,message,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) flags (long int) A flag word controlling some options. See DitsInitiateMessage(3) for full details. In addition to options accepted by DitsInitiateMessage(3), you can specify `DITS_M_SDS`. Set this if the shared memory item contains an exported `SDS` item. If set, the target task can access the item as and argument using `SDS` in the normal way. (see DitsGetArgument(3). If set, this flag causes the target task to pass pass the shared memory address to the routine SdsAccess(3). If false, the shared memory contains data in an application private format and `DRAMA` makes no attempt to interpret it.

(>) path (DitsPathType) Path to the task involved.

(>) SharedMemInfo (DitsSharedMemInfoType *) A structure describing the shared memory section containing the bulk data to be sent. This is set up with the routine DitsDefineShared(3) You should not delete this shared memory until you receive a response to this transaction with code `DITS_REA_BULK_TRANSFERRED` or `DITS_REA_BULK_DONE`.

(>) NotifyBytes (int) If non-zero, it indicates your action should be notified (using `DITS_REA_BULK_TRANSFERRED` entries) every time the specified number of bytes are transferred. Note the this effect is somewhat dependent on the target task behavious and you may only receive the finial `DITS_REA_BULK_DONE` message.

(<) transid (DitsTransIdType *) Transaction id for the transaction started. See DitsInitiateMessage(3) for full details.

(!) message (DitsGsokMessageType *) The message details. See DitsInitiateMessage(3) for full details.

(!) status (StatusType *) Modified status.

Include files: DitsBulk.h

See Also: DitsInitiateMessage(3), DitsInitMessPtr(3), DitsTriggerBulk(3), DitsGetEntReason(3), DitsGetEntBulkInfo(3), DitsPutActions(3), DitsArgIsBulk(3), DitsBulkArgRelease(3), DitsDefineShared(3), DitsDefineSdsShared(3), DitsBulkArgInfo(3), DitsGetArgument(3), DitsGetEntStatus(3).

Support: Tony Farrell, `AAO`

C.77 DitsInitiateMessage — Send a message to another task given a path to that task.

Function: Send a message to another task given a path to that task.

Description: General message sending routine. All of the standard `DRAMA` user level message sending routines (DitsObey etc.) call this routine to send the message.

The following message types can be sent

DITS_MSG_KICK	Kick an action
DITS_MSG_OBEY	Start an action
DITS_MSG_GETPARAM	Get a parameter value
DITS_MSG_MGETPARAM	Get multiple parameters. The name item is ignored (supply as a zero length string). You supply the names of the parameters the values of which are to be returned in the argument structure as a character array containing the space separated names or a set of character string arguments.
DITS_MSG_SETPARAM	Set a parameter value
DITS_MSG_CONTROL	Dits control message
DITS_MSG_MONITOR	Dits parameter monitor message

This is the lower level routine called by routines such as `DitsObey`, to actually send the message. Routines such as `DitsObey` provide a simpler interface, but the use of this routine can be more efficent when the same message is being sent repeatly. In this case, setup the message structure only once to save time.

The control type messages are designed to perform special functions in the Dits fixed part. The following names are currently defined

`DEFAULT`	Set and return the default directory. The argument, if supplied, contains a string in its first item, which is the name of the requested default directory. The new default is returned in the completion message. If the argument is not supplied, then the current default directory is returned.
`MESSAGE`	Requires an argument which should be an message code, as per the Mess routines. The text associated with the message code is fetched and returned in the completion message. The idea of these message is to provide a general facility for translating message codes in the task which knows about the message codes. `DEBUG ->` Sets the `DITS` internal logging level. If no argument is supplied, then level `DITS_LOG_BASIC` is turned on. Otherwise, the argument should be an integer value which is a mask of the values which may be supplied to DitsSetDebug(3). `DUMPPATHS ->` Dump details of all paths used by this task to stdout. `DUMPTRANSIDS ->` Dump details of all outstanding transaction ids known to this task to stdout. `DUMPMON ->` Dump details of current parameter monitors to stdout. `DUMPACTACTIVE ->` Dump details of active actions to stdout. `DUMPACTALL ->` Dump details of all actions to stdout. `VERSIONS ->` Output DRAMa, `DITS` and `IMP` version information. `LOGNOTE ->` If we have a logging system - write the argument (which should be string in an item named Argument1), to the log file. `LOGFLUSH ->` If we have a logging system and it supports flushing, flush it. `LOGINFO ->` Reports details of the logging system, iff a logging system has been enabled and supports this call. `SDSLEAKCHK ->` Allocates and reports the number of an `SDS ID`. This provides a way of working out if a task is leaking `SDS ID`'s. If you run this command a number of times, during which you don't expect any `SDS ID`'s to have been allocated long term, but this number increases, then your task is leaking `SDS ID`'s.

Sending a control message with an invalid name will cause the names of the valid names to be printed on stderr.

The monitor message types are designed to perform parameter monitoring. The following

names are currently defined

`START`	Start monitoring. A monitor id is returned in a trigger message with status `DITS__MON_STARTED` which should be in future messages. The optional argument is a list of parameter names specifing the parameters to be monitored.
`FORWARD`	Start monitoring but forward the details of parameter changes with an obey message to the task specified in the first argument, using the action name specified in the second argument. A monitor id is returned as per `START` and an optional list of parameter names can be specified as additional parameters.
`ADD`	Add the parameters who's names are listed in the second argument onward to the list of parameters being monitored. The first argument is a monitor id as returned from a start or forward message.
`DELETE`	Delete the specified parameters from the list of those being monitored. The first argument is a monitor id as returned from a start or forward message.
`CANCEL`	Cancel monitoring . The argument is the monitor id for which monitoring is being canceled.

Once non-forward monitoring is started, trigger messages will be recevied for each parameter change, in the context of the `START` message transaction. The status will be `DITS__MON_CHANGED` and the argument will be an Sds item of the same name as the parameter being changed and containing the new value.

For forward style monitor, an obey message is sent to the specified task, using the specified action name, each time the parameter value changes.

Monitoring only works if the target task's has a parameter system and that parameter system has enabled monitoring. Each parameter can be monitored by any number monitor transactions.

The parameter name "`_ALL_`" is special and indicates that all parameters are to be monitored. Note that this is incomplatible with the `DITS_M_SENDCUR` flag. If you want to monitor all parameters and fetch their current values, use a parameter get message with the name `_ALL_` to get the initial values.

If when trying to send a monitor message, the buffers in the target task (the parent task for normal monitors and the destination task in forward messages) overflow, then the message is not sent. However, except when a transaction is explictly monitoring all parameters (using the `_ALL_` name), the standard monitor software will request to be notified when the buffers in the target task empty. When the notification comes in, It will then send a

monitor message giving the value of the parameter at that time. This ensures last value of the parameter gets though, regardless of buffers being overflowed.

For `GET` and `MGET` messages, the names `_ALL_` and `_NAMES_` are special. If `_ALL_` is specified, the entire parameter system is returned. If `_NAMES_` are specified it requests that a list of names be returned. In this case an `SDS` structure of type is returned. This will contain one item, an array with each of the names in it. If you wish to get a parameter with a name that is longer then `DITS_C_NAMELEN` (the length of `message->name`) then you should use a `MGET` message and specify the name in the argument structure.

For `SET` messages, if the name is to be longer then `DITS_C_NAMELEN`, then specify the name `_LONG_` and then specify 2 items in the `SDS` message.argument structure. The first of these should be the a character string giving the name of the parameter and the second should be the value.

If the action you are obeying is spawnable (see DitsPutActions(3)) then kicking is requires a special kick message. There are two alternatives, the first is to use the transid value returned in the obey call to create an argument structure to be supplied to the kick message, This is done using DitsSpawnKickArg(3) or DitsSpawnKickArgUpdate(3). Other details can be added to this structure if required and will be available to the kick with DitsGetArgument()- through an item named KickByTransId should be ignored by application code.

Alternatively, if the action index of the started action may used, by specifying an argument structure to the message containing an item named KickByIndex and which contains this acton index value. The subsidiary action can use DitsGetActIndex(3) to gets its action index. How the value gets form the subsidiary task to the one which wants to sent the message is up to the application.

Unless transid is null, the sender should reschedule (or block using DitsActionWait(3)/DitsUfaceWait(3) to await responses. See DitsGetEntReason(3) for a list of possible entry reasons when responses arrive (not all the values will occur due to responses from sending a messages).

If a response has an argument (`SDS`) associated with it, the normally DitsGetArgument(3) can be used to get the argument. Action (Obey) complete messages would have put that argument using DitsPutArgument(3).

The status of a response message can be returned using DitsGetEntStatus(3), when the reaponse is received. Of particular note are the following status codes.

DITS__ACTACTIVE	An obey message was sent, but the target action is already active and is not a spawnable action.
DITS__COMPSENDERR	A message that was to be sent with the completion message is too big to ever fit in the buffer. The task which set up the buffer should increase the buffer sizes, or reduce the size of any argument to the message.
DITS__NOSPACE	The buffer for the reply does not have sufficent space for this message at this time.
DITS__ACTNOTACT	An attempt was made to kick an action which is not currently active.
DITS__NOKICK	An attempt was made to kick an action which has no kick handler.
DITS__TASKDISC	A task disconnected (died) whist processing the message.
DITS__MACHLOST	All connections to the machine on which the task is running have been lost.
DITS__MON_STARTED	Indicates a monitor transaction has started.
DITS__MON_CHANGED	Indicates a monitor transaction parameter value changed message.

Language: C

Call:

(Void) = DitsInitiateMessage (flags, path, transid, message, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) flags (long int) A flag word controlling some options. The priority bit (`DITS_M_PRIORITY`) can be set to indicate that this message should be sent to the top of the queue of messages being sent to the target task. Note that only relatively short messages should be sent in this way , as the Imp priority buffers are limited in size. (It is recommend that the message not include an argument, in order to meet this restriction.

(>) path (DitsPathType) Path to the task involved.

(<) transid (DitsTransIdType *) Transaction id for the transaction started. If an address of zero is supplied, then don't create a transaction id. In this which case we will never be notified of errors one the message is sent. As a special case, if you specify a transid of 0 when sending a Kick message, then any Informational/Error or Trigger messages initiated during the handling of the initial Kick message will be directed towards the initiator of the Corresponding Obey message.

(!) message (DitsGsokMessageType *) The message details. The following detail should be set

flags	(int) Mask of flags. Set `DITS_M_ARGUMENT` if an argument is supplied. Set `DITS_M_SENDCUR` if a monitor message (`START`/`FORWARD`/`ADD`) and you want the current value of the parameter sent immediately. Set `DITS_M_REP_MON_LOSS` to cause the reporting of monitor messages which are lost due to waiting for buffer empty notification messages to arrive. This should be consider if the loss of monitor messages is significant to your application. In general, it is not as the system ensures the last parameter update gets though.
argument	(SdsIdType) The Sds id of the argument. Ignored if flags is not set to `DITS_M_ARGUMENT`. For actions etc. this value will be available to using DitsGetArgument().
type	(DitsMsgType) The message type. As listed above.
name	(DitsNameType) The name of the action to initialite or the parameter to set. Set this value using the macro "DitsNameSet(destin,source)". where source is the name you wish to setup and destin is the field.

Other fields may be modified during the send, so this routine should be able to write to this location.

(!) status (StatusType *) Modified status.

External values used: DitsTask

Prior requirements: Can only be called from a Dits application routine or a user interface response routine

See Also: The Dits Specification Document, DitsObey(3), DitsKick(3), DitsSetParam(3), DitsGetParam(3), DitsPathGet(3). DuiExecuteCmd(3), DulMessageW(3), DitsActionWait(3), DitsUfaceWait(3), DitsGetArgument(3), DitsPutAction(3), DitsGetEntReason(3).

Support: Tony Farrell, `AAO`

C.78 DitsInterested — Indicates the current action is interested in certain messages.

Function: Indicates the current action is interested in certain messages.

Description: An action may be caused to reenter when external messages are received from subsidiary actions. This routine indicates additional message types the action is interested in.

By default, the action will be interested in trigger, transaction failure and transaction completion messages

Note, this function is implemented as a macro.

Language: C

Call:
(Void) = DitsInterested(mask, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) mask (DitsMsgMaskType) A bit mask indicating which additional messages the action is interested in. Any of the following bits may be set

`DITS_MSG_M_ERROR`	Error reports, those messages sent by a call to ErrOut or ErrFlush.
`DITS_MSG_M_MESSAGE`	Messages to the user interface, those messages sent by MsgOut.
`DITS_MSG_M_TRIGGER`	Trigger messages, those messages sent by DitsTrigger.
`DITS_MSG_M_TRANSFAIL`	Transaction failure messages
`DITS_MSG_M_COMPLETION`	Transaction completion messages

(`!`) status (StatusType `*`) Modified status.

Include files: DitsInteraction.h

External functions used: None

External values used: TheTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine

See Also: The Dits Specification Document, DitsNotInterested, DitsInitiateMessage(3).

Support: Tony Farrell, `AAO`

C.79 DitsIsActionActive — Indicates if the specified action is active.

Function: Indicates if the specified action is active.

Description: You supply an action index and this routine indicates if that action is currently active.

Language: C

Call:
(void) = DitsIsActionActive (index, active, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) index (long int) The index to the action. See DitsActIndexByName() or DitsActIndexByName.

(<) active (int *) Returns true of false to indicate if the action is currently active.

(!) status (StatusType *) Modified status.

Include files: DitsUtil.h

External functions used:

External values used: DitsTask

Prior requirements: Can only be called from a Dits Application routine

See Also: The Dits Specification Document, DitsKillByIndex DitsSignalByIndex(3), DitsGetActIndex(3), DitsActIndexByName(3).

Support: Tony Farrell, `AAO`

C.80 DitsIsOrphan — Indicate if a transaction is an orphan.

Function: Indicate if a transaction is an orphan.

Description: Returns true if the specified transaction has been made an orphan by the action which initiated it, either explictly by DitsForget or implictly by the action returning before the transaction completes.

This routine can be used by Orphan handling actions to distingish orphans from transacations started by the action.

Language: C

Call:
(int) = DitsIsOrphan (transid)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) tranid (DitsTransIdType) The transaction to check.

Include files: DitsOrphan.h

External values used: DitsTask

Prior requirements: DitsAppInit must have been called

See Also: The Dits Specification Document, DitsCheckTransactions(3), DitsTakeOrphans(3), DitsPutOrphanHandler(3), DitsForget(3).

Support: Tony Farrell, `AAO`

C.81 DitsIsRegTestMode — Returns true if `DRAMA` running in regression test mode.

Function: Returns true if `DRAMA` running in regression test mode.

Description: This returns true if the environment variable `DRAMA_REG_TEST` is defined. Programs can use it to ensure their output is compatible with regression testing (e.g. don't out floating point values with more precision then is stable across machines don't output fine timing information which is machine performance specific.

Language: C

Call:
(int) = DitsIsRegTestMode ()

Include files: DitsUtil.h

External functions used:

DitsGetSymbol		

External values used: None

Prior requirements: None

Support: Tony Farrell, `AAO`

C.82 DitsKick — Send a KICK message to a task

Function: Send a KICK message to a task

Description: A KICK message with the specified action name is sent to the task on the given path. The specified argument is put in the message.

A kick message will only be accepted if the action to which it is directed is already active. The action does not have to be a subsidiary of this action or any other action in this task. A kick message causes the target action's kick routine to be invoked.

If the KICK fails, a "transaction failure" message will be received by the task with the transaction id returned by this routine. Otherwise a completion message will be received when the task has responded and the current action should reschedule to await this.

As a special case, if you specify a transid of 0 when sending a Kick message, then any Informational/Error or Trigger messages initiated during the handling of the initial Kick message will be directed towards the initiator of the Corresponding Obey message.

Note, this function is implemented using a macro to invoke DitsInitiateMessage.

Language: C

Call:

(Void) = DitsKick(path,action,argument,transid, status)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(>) path (DitsPathType) The path to the task to which the message is directed. See DitsGetPath for details about how to obtain a path to a task.

(>) action (char *) The null terminated name of the action to kick.

(>) argument (SdsIdType) An argument to the action. This should be an Sds id. See DitsGetArgument and DitsPutArgument for more details on action arguments.

(<) transid (DitsTransIdType *) Transaction id for the transaction started. If an address of zero is supplied, then don't create a transaction id. In this case we will never be notified of errors once the message is sent.

(!) status (StatusType *) Modified status.

Include files: DitsInteraction.h

External functions used:

DitsInitiateMessage	Dits internal	Send a message to a task.

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or when the context if DITS_CTX_UFACE.

See Also: The Dits Specification Document, DitsInitiateMessage(3), DitsObey(3), DitsSetParam(3), DitsGetParam(3), DitsPathGet(3), DuiExecuteCmd(3), DulMessageW(3)

Support: Tony Farrell, AAO

C.83 DitsKillByIndex — Kill a currently active action.

Function: Kill a currently active action.

Description: The routine allows an action to kill another action. The other action will complete immediatly without rescheduling. An action completion message is sent to the originator of the action being killed, with status set to the status specified

Note that an error will occur if an action attemps to kill itself (Status will be set to DITS__CANTKILLSELF).

Note, actions which are blocked using DitsActionWait() or DitsActionTransIdWait() will be worken up and the call to DitsActionWait()/DitsActoinTransIdWait() will return ither the killstat or, if that is STATUS__OK, DITS__WAIT_ACT_KILLED. The action is not killed immediately and there is actually some scope for the kill to be prevented in this case (by the action implementation hanlding the status, but that is considered rude).

Language: C

Call:
(Void) = DitsKillByIndex (name, argument, killstat, status)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(>) index (long int) The index to the action to kick. This must be a value returned by DitsGetActIndex().

(>) argument (SdsIdType) An argument to return with the action completion message. This should be an Sds id. See DitsGetArgument for more details on action arguments.

(>) killstat (StatusType) The status of the completion message to be sent to the originator. DITS__ACTKILLED may be specified if no particular status is required.

(!) status (StatusType *) Modified status.

Include files: DitsSignal.h

External functions used:

Dits___SendTap	Dits internal	Send a tap message
Dits___Orphaned	Dits interal	Tidy up outstanding transactions at the end of an obey.

External values used: DitsTask

Prior requirements: Can only be called from a Dits Application routine or a user interface response routine

Support: Tony Farrell, `AAO`

See Also: The Dits Specification Document, DitsKillByName(3), DitsSignalByIndex(3), DitsGetActIndex(3)

C.84 DitsKillByName — Kill a currently active action.

Function: Kill a currently active action.

Description: The routine allows an action to kill another action. The other action will complete immediatly without rescheduling. An action completion message is sent to the originator of the action being killed, with status set to that specified.

Note that an error will occur if an action attemps to kill itself (Status will be set to `DITS__CANTKILLSELF`).

Note, actions which are blocked using DitsActionWait() or DitsActionTransIdWait() will be worken up and the call to DitsActionWait()/DitsActoinTransIdWait() will return ither the killstat or, if that is `STATUS__OK`, `DITS__WAIT_ACT_KILLED`. The action is not killed immediately and there is actually some scope for the kill to be prevented in this case (by the action implementation hanlding the status, but that is considered rude).

For spawable actions, all actions of the same name which are active are killed.

Language: C

Call:
(Void) = DitsKillByName (name, argument, killstat, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) name (Char *) The null terminated name of the action to kill.

(>) argument (SdsIdType) An argument to return with the action completion message. This should be an Sds id. See DitsGetArgument for more details on action arguments.

(>) killstat (StatusType) The status of the completion message to be sent to the originator. `DITS__ACTKILLED` may be specified if no particular status is required.

(!) status (StatusType *) Modified status.

Include files: DitsSignal.h

External functions used:

Dits___ActptrByName	Dits internal	Return the index to an action
DitsKillByIndex	Dits	Kill an action given the index.

External values used: DitsTask

Prior requirements: Can only be called from a Dits Application routine or a user interface response routine

See Also: The Dits Specification Document, DitsKillByIndex(3), DitsSignalByName(3), DitsGetActIndex(3)

Support: Tony Farrell, `AAO`

C.85 DitsLoad — Loads and runs a task.

Function: Loads and runs a task.

Description: This routine is used to request that `DITS` load and run a new task. The calling routine specifies the task to be run and the machine on which it is to be run. This routine initiates a transation which uses the Imp Master task on the target machine to load the task. This means that the the imp master task must be running on the target machine. The imp master task is normally started with the dits_netstart command, see Dits documentation for details.

Reschedules resulting from initiating this transaction are

`DITS_REA_LOAD`	The task has been loaded successfully and has registered with the message system. `DitsPathGet()` should now succeed. Note, the transaction has not completed.
`DITS_REA_LOADFAILED`	The load has failed and the transaction is complete.
`DITS_REA_EXIT`	The loaded task has exited. This message indicates the program has then exited for whatever reason. The transaction is now complete.

Is it possible to load programs which are not Dits programs. In this case, `DITS_REA_LOADED` will never be received.

Language: C

Call:

void = DitsLoad (Node, Program, ArgString, Flags, TaskParams, transid, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) Node (char *) Specifies the machine on which the task is to run. This can be any string that can interpreted by the underlying networking system, for example a symbolic or numeric `IP` address. If this parameter is a `NULL` address, or points to a null string the current machine is used.

(>) Program (char *) Specifies the program to be run in the new task. See below for a more detailed description of how a program may be specified.

(>) ArgString (char *) Arguments to be passed to the loaded task, formatted into a single character string. See below for a more detailed discussion of task arguments.

(>) Flags (long int *) A flag word controlling some options. In general these options are system-dependent, at least to some extent. The various options are listed below.

(>) TaskParams (DitsTaskParamType *) Address of a structure containing additional values used to control the loading of the task. The use of the various fields in this structure is controlled by the setting of the bits in the Flags argument.

(<) transid (DitsTransIdType *) Transaction id for the transaction started. If an address of zero is supplied, then don't create a transaction id. In this case, we will never be notified of load events.

(!) status (Int *) Modified status.

TaskParams fields to be set prior to the call: Priority(int), Bytes(int), ProcName(char [32]), Decode(char[64])

Program syntax: Generally, what is specified as 'Program' will be a string indicating an executable file. In the case of systems such as VxWorks, where programs are pre-loaded subroutines, a separate mechanism is provided to associate strings that can be specified as 'Program' with the routines to be run. In such a case, any string can be associated with a subroutine. In larger systems such as `VMS` or `UNIX`, where the name has to indicate a file, questions of filename syntax arise. In all cases, 'Program' can specify an explicit file in the syntax accepted by the target machine, eg '`SYSDISK1:[KS.PROGS]FRED`' for `VMS` ('`.EXE`' being assumed), or '/home/aaossc/ks/progs/fred' for `UNIX` (no extension assumed). In most cases, programmers will prefer to use some more flexible way of specifying the file. So under `VMS` one could use '`EXE_DIR:FRED`' where `EXE_DIR` is a logical name pointing to the directory containing `FRED.EXE`. `UNIX` has no such built-in facility, but to provide the required flexibility the `IMP` system under `UNIX` will interpret the string 'exe_dir:fred' as meaning the program 'fred' in a directory equated to an environment variable (as defined for the IMP_Master task) called 'exe_dir'. Note that `IMP` passes such names to the underlying system without any changes to case. This means that the string like 'exe_dir:fred' could be understood by `IMP` systems running on all of `VMS`, `UNIX` and VxWorks. A string such as 'fred' will be taken under `UNIX` to refer to a file called 'fred' in the `PATH` as defined for the IMP_Master task. Under `VMS`, 'fred' will be taken to be either a file called 'fred.exe' in the default directory for the IMP_Master task. Alternatively, under `VMS` - again to provide increased flexibility - 'fred' can be defined as a symbol causing a program to be run, so long as this symbol is defined for the IMP_master task. If parameters are to be passed directly to a task run under `VMS`, it has to be run in this way.

In summary, it is possible to set up the IMP_master task on all systems so that a simple name, eg 'fred' can be used as the taskname, or so that a name apparently involving a symbolic reference to a directory eg 'exe_dir:fred' can be used. Alternatively, for systems that have file systems, a full system-specific file specification can be provided.

Argument strings: The argument string is a character string containing an argument string to be passed to the task to be run. In many cases this is expected to be null, the message

system being used to communicate with the task. In some cases, however, particularly where a general purpose program (one that does not use `IMP`) is being run, the argument string can be used to pass arguments to it. The `IMP` system will pass the argument string on to the task in as unchanged a form as possible, but this will in practice vary with the system in use. Under `VMS`, the string can be passed to the program unchanged, so for example, 'Program' could be '`DELETE`' and 'ArgString' could be '`/NOCONFIRM SCRAP.DAT;*`'. Under `UNIX` the string will be split into separate arguments (separated by spaces, allowing for strings in double or single quotes) and these passed to the task. So under `UNIX` 'Program' could be 'rm' and 'ArgString' could be '-i scrap.dat'. Note that `IMP` will not interpret characters such as '`*`', as a `UNIX` shell would. Systems such as VxWorks where the task that is run is a subroutine that has to be called directly will normally have the argument string passed as a single character string. If the string is to be interpreted as a set of numeric and/or character values, the `DITS_ARG_SPLIT` flag should be specified - see below.

Possible flag values: The following bit masks are defined in DitsInteraction.h, and have the following effects if used to set bits in the Flags argument:

`DITS_M_ARG_SPLIT` For VxWorks type tareget systems only. TaskParams.Decode should be a string similar to that used by printf(), containing %d, %f and %s codes to indicate how the argument string is to be interpreted in order for the arguments to be passed to the task.

`DITS_M_REL_PRIO` If specified, TaskParams.Priority will be an integer used to set the priority of the loaded task relative to the priority of the IMP_Master task. A positive number will give the loaded task a priority that is higher than that of the master task, a negative number will give a lower priority. (The master task may be limited in the priority it can assign to a loaded task, particularly if asked to run it at high priority. Normally, tasks are loaded at the same priority as the master task.

`DITS_M_ABS_PRIO` If specified, TaskParams.Priority will be an integer used to set the priority of the loaded task in an absolute way. To use this the caller will have to know the priority conventions used by the target system.

`DITS_M_SYMBOL` Used where the target system is a `VMS` system to cover any ambiguity in the interpretation of a 'Program' string as either the actual name of a program to be executed or a symbol that causes a program to be executed. If this is set, it forces the string to be interpreted as a symbol.

`DITS_M_PROG` Used where the target system is a `VMS` system to cover any ambiguity in the interpretation of a 'Program' string as either the actual name of a program to be executed or a symbol that causes a program to be executed. If this is set, it forces the string to be interpreted as a program name. (The difference is that a program name needs to be run using a spawned `RUN` command, or - if no arguments are to be passed to it, by a direct call the the `VMS $CREPAR` system routine.)

`DITS_M_NAMES` Used to control the inheritance of symbols (logical names and symbols under `VMS`, environment variables under `UNIX`) by the new task. If specified, this flag insists that symbols known to the IMP_Master task be inherited by the new task. If not set, it is at the discretion of the master task whether this is done.

`DITS_M_SET_BYTES` If specified TaskParams.Bytes is a byte count value used to specify the

amount of memory to be used by the task. This is currently only supported by VxWorks, where it is used to specify the stack size for the task. If not specified, a default size will be used.

Other flags will be included as they prove to be needed.

TaskParam fields: The TaskParam structure supplies the following fields. Not all are needed in all cases, but ProcName should normally be set. The rest depend on the settings of the various Flags bits.

TaskParam.ProcName is a character string used to supply a name for the task to be created. Not all systems support giving names to individual processes, but those that do usually insist that one be supplied and that it be unique. Note that this is a 'system' name, used by the operating system, and is not the name under which the loaded task will register with the `IMP` system (assuming it does so).

TaskParam.Decode is an argument decoding specification string used only if the `DITS_ARG_SPLIT` flag is set in Flags. See the description of this flag for more details.

TaskParam.Priority is a priority specification used only if either the `DITS_ABS_PRIO` flag or the `DITS_REL_PRIO` flag is set in Flags. See the description of these flags for more details.

TaskParam.Bytes is a byte count value used to specify the amount of memory to be used by the task. At present this is only supported by VxWorks where it specifies the task size for the task.

Include files: DitsInteraction.h

External functions used:

ImpRunTask	Imp	Run a task

External values used: DitsTask = Details of the current task.

Prior requirements: Should only be called from a Dits application routine or when the context is `DITS_CTX_UFACE`.

See Also: The Dits Specification Document, DitsLoadErrorStat(3), DitsLoadErrorText(3), DitsGetEntInfo(3), DitsAppInit(3), DulLoadW(3), Imp manual.

Support: Tony Farrell, `AAO`

C.86 DitsLoadErrorStat — Returns the system status code associated with a load error.

Function: Returns the system status code associated with a load error.

Description: When a load operation fails or a task which had been loaded exits, the system status is returned. This function fetches this value. This function is only sensible if DitsGetReason has returned a reason of `DITS_REA_LOADFAILED` or `DITS_REA_EXIT`.

In addition, this value may be hard to interpret is the load was on a remote machine, so DitsLoadErrorText may be more appropiate.

Note, this function is implemented as a macro.

Language: C

Call:
(long int) = DitsLoadErrorStat()

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output) None

Include files: DitsInteraction.h

Function value: The status code.

External functions used: None

External values used: TheTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or user interface response routine.

See Also: The Dits Specification Document, DitsLoad(3), DitsGetEntReason(3), DitsLoadErrorText(3).

Support: Tony Farrell, `AAO`

C.87 DitsLoadErrorText — Returns the system status code text associated with a load error.

Function: Returns the system status code text associated with a load error.

Description: When a load operation fails or a task which had been loaded exits, the system status is returned. This function fetches a text representation of this code as translated by the machine on which the load was preformed. This function is only sensible if DitsGetReason has returned a reason of `DITS_REA_LOADFAILED` or `DITS_REA_EXIT`.

Note, this function is implemented as a macro.

Language: C

Call:
(char `*`) = DitsLoadErrorText()

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output) None

Include files: DitsInteraction.h

Function value: The address of a null termiante string, read only.

External functions used: None

External values used: TheTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or user interface response routine.

See Also: The Dits Specification Document, DitsLoad(3), DitsGetEntReason(3), DitsLoadErrorStat(3).

Support: Tony Farrell, AAO

C.88 DitsLogFlush — Flush the log file.

Function: Flush the log file.

Description: If DRAMA has an enabled logging system (see DitsSetLogSys()) then this routine calls the logFlush routine enabled by that logging system. The intention is to allow libraries and user code to flush the log file, if appropite.

Language: C

Call:
(void) = DitsLogFlush (status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(!) status (StatusType *) Modified status.

Include files: DitsUtil.h

External functions used:

External values used: DitsTask

Prior requirements: Can only be invoked after DitsAppInit().

See Also: The Dits Specification Document. DitsSetLogSys(3).

Support: Tony Farrell, AAO

C.89 DitsLogMsg — Write a message to the log file.

Function: Write a message to the log file.

Description: If DRAMA has an enabled logging system (see DitsSetLogSys()) then this routine calls the logLogMsg routine enabled by that logging system. The intention is to allow libraries and user code to log, without having to know what logging system is enable, if any.

Language: C

Call:
(void) = DitsLogMsg (level, prefix, status, message,...)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(>) level (unsigned) A logging level indicator. One of the values below should be set to indicate what is being logged such that the logging system can select what is actually formated and written as desired.

DITS_LOG_STARTUP	Message is a startup mesage.
DITS_LOG_INST	Message concerns running instrumentation
DITS_LOG_USER1	User defined.
DITS_LOG_USER2	User defined.
DITS_LOG_ALL	Always log this message to the file.

(>) prefix (char *) A prefix string for the message, indicating say the library concerned.

(!) status (StatusType *) Modified status. (>) message The message as a C-Printf style format string. Subsequent arguments are arguments to the printf format.

Include files: DitsUtil.h

External functions used:

External values used: DitsTask

Prior requirements: Can only be called from a Dits Application routine

See Also: The Dits Specification Document. DitsSetLogSys(3).

Support: Tony Farrell, AAO

C.90 DitsLosePath — Lose a path.

Function: Lose a path.

Description: Disconnect from another task. This routine marks a path as invalid and disconnects from the task.

A future call to DitsGetPath will now use the underlying network to find the task.

An exception is made if the path in question is a path to the calling path (the self path). In this case, then underlying `IMP` connection is not closed and the disconnect handler is never invoked. But all other cleaning up appropiate for a disconnection of the path is still done. This is necessary to ensure the selfPath continues to behave as it normally would.

If there is currently any outstanding GetPath transaction on this path, you should consider what is to happen to them. The actions/uface handlers concerned will receive messages with reasons of `DITS_REA_DIED` and status of `DITS__TASKDISC`.

If your intention is to re-connect to the task involved, you should wait at least a second (longer if slow network connections are involved) using an action wait or uface timers. This is required to allow the processing through the `IMP` layer of the various messages involved. If you fail to do this, you may find that your GetPath operation fails.

If there are currently any outstanding notify request transactions, again you should consider making them orphans.

In this version's implementation, any notify request transactions, (which include transactions set up by DitsPathGet if we already have a path to task) will never complete after this call. A future version of this software may change this.

(In previous versions of `DRAMA`, prior to V 1.0, there would only be one such transaction could have existed and it was automatically deleted. From `V1.0`, there may be many such transactions so it is left up to the user to determine what to do with them If the GetPath operation completes after this call, the transactions will be trigged but the path will be unuseable.)

Language: C

Call:
(Void) = DitsLosePath(path,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) path (DitsPathType) The path to lose.

(!) status (StatusType *) Modified status. Possible failure codes are

Include files: DitsInteraction.h

External functions used: ImpCloseConnect

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine

See Also: The Dits Specification Document, DitsPathGet(3).

Support: Tony Farrell, AAO

C.91 DitsMainLoop — Dits main loop

Function: Dits main loop

Description: This is the main message loop routine. It blocks waiting for messages and processes them when they are received. It does this until an application routine requests that the task exit using DitsPutRequest.

Assuming there is no internal error in Dits, the status value returned by this function is the completion status of the user action routine with requested the exit.

Note, this function is implemented as a macro.

Language: C

Call:
(Void) = DitsMainLoop (status)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used:

DitsMsgReceive	Dits	Receive a message.

External values used: DitsTask

Prior requirements: DitsAppInit should have been called. This routine is NOT to be called from user action routines, but instead from the user main program.

See Also: The Dits Specification Document, DitsMsgReceive(3), DitsMsgAvail(3), DtclAppMainLoop(3), DtclAppTkMainLoop(3), DitsAltInLoop(3).

Support: Tony Farrell, AAO

C.92 DitsMonitor — Called by a parameter system to notify of a parameter value change

Function: Called by a parameter system to notify of a parameter value change

Description: See if this parameter is known to the monitor system. If so, lookup the details of the paths and transaction id's involved in all monitors. Send tap messages for each normal style monitor and Obey messages for forward style monitors.

Language: C

Call:
(Void) = DitsMonitor (parName, flag, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) parName (Char *) The parameter name

(>) flag (DitsMonitorFlagType) Indicate how the value item can be treated. Possibilites are-

`DITS_MON_TRUST`	We can trust that the name of the sds item is the same as the parameter name. This is the most efficent technqiue.
`DITS_MON_NOTRUST`	Don't trust the name as being correct. We will copy the value to a new Sds structure and give it the correct name.
`DITS_MON_CHANGE`	Don't trust the name as being correct but it is ok to rename it.

(>) value (SdsIdType) The Sds id of the new parameter value. This is sent as is, except as indicated by the flag argument.

(!) status (StatusType *) Modified status.

Include files: DitsMonitor.h

External functions used:

DitsNameSet	Dits	Set a name type.
Dits___MonitorParam	Dits	Get parameter details.
Dits___SendTap	Dits	Send a tap message.
SdsRename	Sds	Rename a sds item.
SdsCopy	Sds	Copy an sds item.
SdsDelete	Sds	Delete an sds item.
SdsFreeId	Sds	Free and sds item.

External values used: DitsTask

Prior requirements: DitsMonitorInit should have been called.

See Also: The Dits Specification Document, DitsAppParamSys(3), DitsMonitorDisconnect(3), DitsMonitorMsg(3), DitsMonitorTidy(3), SdpInit(3).

Support: Tony Farrell, AAO

C.93 DitsMonitorDisconnect — A task with which this task was communicating has disconnected.

Function: A task with which this task was communicating has disconnected.

Description: Go through the list of monitor transaction structures to find all transactions along the given path. Delete each transaction.

Language: C

Call:
(Void) = DitsMonitorDisconnect (path, status)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(>) path (Dits___PathType *) The path which is disconnecting.

(!) status (StatusType *) Modified status.

Include files: DitsMonitor.h

External functions used:

ErsOut	Ers	Output error messages
Dits___SendTap	Dits internal	Send a tap message

External values used: DitsTask - details of the current atsk.

Prior requirements: DitsAppInit must have been called.

See Also: The Dits Specification Document, DitsAppParamSys(3), DitsMonitor(3), DitsMonitorMsg(3), DitsMonitorTidy(3), SdpInit(3).

Support: Tony Farrell, AAO

C.94 DitsMonitorMsg — Handle incomming monitor messages

Function: Handle incomming monitor messages

Description: There five monitor messages.

`START`	Start a monitor transaction. Creates a transaction struct and return the id in a tap message. Then preform a `ADD` operation with the remaining arguments.
`ADD`	Add new parameters to those being monitored.
`DELETE`	Remove parameters from list of those being monitored
`CANCEL`	`DELETE` all parameters and cancel the monitor transaction.
`FORWARD`	Start a monitor transaction that will cause parameter change messages to be forwarded to anothr task. The `TASK` and `ACTION` to be used are specified and then and `ADD` is done on the remaining arguments.

Language: C

Call:

(Void) = DitsMonitorMsg (flags, name, argin, transid, path, complete, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) flags (`INT32`) If the bit `DITS_M_SENDCUR` is set, then for message types `START`/`FORWARD`/`ADD`, we send the initial value of the parameter immediately.

(>) name (char *) The name of the message, see above.

(>) argin (SdsIdType *) The input argument

(>) transid (Dits___NetTransIdType) The transaction of incomming message.

(>) path (Dits___PathType) The path of the incomming mesasge

(>) tag (long int) The tag associated with the orignal message.

(<) complete (int *) Set true to indicate the transaction is complete.

(!) status (StatusType *) Modified status.

Include files: DitsMonitor.h

External functions used:

strcmp	CRTL	Compare two strings
malloc	CRTL	Allocate memory
ArgNew	Arg	Create an argument
ArgPuti	Arg	Put an integer item.
ArgDelete	Arg	Delete an argument structure.
ArgCvt	Arg	Convert an argument
Dits___SendTap	Dits internal	send a tap message
DitsPathGet	Dits	A a path to a task
SdsIndex	Sds	Find an item by index

External values used: DitsTask - detail of the current task.

Prior requirements: DitsAppInit must have been called.

See Also: The Dits Specification Document, DitsAppParamSys(3), DitsMonitor(3), DitsMonitorDisconnect(3), DitsMonitorTidy(3), SdpInit(3).

Support: Tony Farrell, AAO

C.95 DitsMonitorReport — Report on all the parameters being monitored in a task

Function: Report on all the parameters being monitored in a task

Description: Walk the linked list of monitored parameters, reporting on which parameters are being monitored, and by whom.

Language: C

Call:
(Void) = DitsMonitorReport (func, data, status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) func (DitsOutputRoutineType) A function to called to output each line. It is passed the data item and must be fprintf compatible.

(>) data (void *) Passed directly to func.

(!) status (int *) Modified status.

Include files: DitsMonitor.h

External values used: DitsTask - details of the current task.

Prior requirements: DitsAppInit must have been called.

Support: Tony Farrell, AAO

C.96 DitsMonitorTidy — Tidy up the monitor code at task shutdown

Function: Tidy up the monitor code at task shutdown

Description: Release memory used in monitor transactions.

Language: C

Call:
(Void) = DitsMonitorTidy (status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

(!) status (!) status (Int *) Modified status.

Include files: DitsMonitor.h

External functions used:

free	CRTL	Free memory

External values used: DitsTask - details of the current task.

Prior requirements: DitsAppInit must have been called.

See Also: The Dits Specification Document, DitsAppParamSys(3), DitsMonitor(3), DitsMonitorDisconnect(3), DitsMonitorMsg(3), SdpInit(3).

Support: Tony Farrell, AAO

C.97 DitsMsgAvail — Returns the count of available messages.

Function: Returns the count of available messages.

Description: If the count of available messages is greater then zero, then a call to DitsMsgReceive will not block. This routine can be called from any action as well as part of a main loop.

An action may want to call this routine to determine whether to reschedule a time consuming process in order to process other messages.

Language: C

Call:
(Unsigned long Int) = DitsMsgAvail (status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

(!) status (StatusType *) Modified status.

Function value: The number of messages available

Include files: DitsSys.h

External functions used:

ImpMessageCount	Imp	Returns the count of available messages.

External values used: DitsTask - Details of the current task

Prior requirements: DitsAppInit should have been called.

See Also: The Dits Specification Document, DitsMsgReceive(3), DitsMainLoop(3).

Support: Tony Farrell, `AAO`

C.98 DitsMsgReceive — Process a message

Function: Process a message

Description: Wait for and processes a message . If necessary it causes user action routines to be invoked. In most cases a User action routine will be invoked in response to the message, although some messages are handled totally in this routine. In the case of a task or machine with which this task was communicating, being lost, several user action routines may be invoked.

Language: C

Call: (Void) = DitsMsgReceive (exitflag, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(<) exitflag (long int *) Returns true if task should exit. This occurs if either a user action routine request the task to exit (using DitsPutRequest) or an Imp shutdown message is received.

(!) status (StatusType *) Modified status. If exitflag is false, then this is only set to indicate a critical dits internal error. If exitflag is true, then it is the value of status returned by the action routine which requested the task exit. If the exit was requested by an Imp shutdown message, then status is set to `DITS__IMPSHUTDOWN`.

Include files: DitsSys.h

External functions used:

Dits___TransIdDelete	Dits internal	Delete a transaction id.
Dits___MsgImpSys	Dits internal	Process Imp system messages.
Dits___MsgDitsMsg	Dits internal	Process Dits messages
Dits___MsgResponse	Dits interal	Response to a message
ImpReadEnd	Imp	Indicates message processing is finished
impReadPtr	Imp	Read a message.
SdsFreeId	Sds	Free an Sds id.
SdsDeleteId	SDs	Delete an Sds item.
ErsRep	Ers	Report error messages
ErsOut	Ers	Output error messages.

External values used: DitsTask - Details of the current task

Prior requirements: DitsAppInit should have been called

See Also: The Dits Specification Document, DitsMsgAvail(3), DitsMainLoop(3).

Support: Tony Farrell, `AAO`

C.99 DitsMsgWait — Wait until a message arrives and return before processing it.

Function: Wait until a message arrives and return before processing it.

Description: This routine will block until a `DRAMA` message is available for the task. If will then return prior to processing that message such that DitsMsgReceive() can then process the message.

This is not required as DitsMsgReceive() will block to wait for the message, but can be useful if you need to do something (take a semaphore) between reception and processing of a message.

This makes use the message peeking facility, so the first message to arrive will be considered peeked at and therefore not accessible to the DitsPeek(3) routine.

Language: C

Call:

(void) = DitsMsgWait (status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`!`) status (StatusType `*`) Modified status.

Include files: DitsSys.h

External functions used:

ImpReadPtr	Imp	Wait for a message.
ImpReadEnd	Imp	Indication completion of read.

External values used: DitsTask - Details of the current task

Prior requirements: DitsAppInit should have been called.

See Also: The Dits Specification Document, DitsMsgReceive(3), DitsMainLoop(3), DitsMsgAvail(3), DitsPeek(3).

Support: Tony Farrell, `AAO`

C.100 DitsNotInterested — Indicates the current action is not interested in certain messages.

Function: Indicates the current action is not interested in certain messages.

Description: An action may be caused to reenter when external messages are received from subsidiary actions. This routine indicates message types the action is no longer interested in. Any messages an action is not interested in are forwarded to the initiator of the action.

By default, the action will be interested in trigger, transaction failure and transaction completion messages

Note, this function is implemented as a macro.

Language: C

Call:

(Void) = DitsNotInterested(mask, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) mask (DitsMsgMaskType) A bit mask indicating which messages the action is no longer interested in. Any of the following bits may be set

`DITS_MSG_M_ERROR`	Error reports, those messages sent by a call to ErrOut or ErrFlush.
`DITS_MSG_M_MESSAGE`	Messages to the user interface, those messages sent by MsgOut.
`DITS_MSG_M_TRIGGER`	Trigger messages, those messages sent by DitsTrigger.
`DITS_MSG_M_TRANSFAIL`	Transaction failure messages
`DITS_MSG_M_COMPLETION`	Transaction completion messages

(!) status (StatusType *) Modified status.

Include files: DitsInteraction.h

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine

See Also: The Dits Specification Document, DitsInterested, DitsInitiateMessage(3).

Support: Tony Farrell, AAO

C.101 DitsNumber — Return the size of an array.

Function: Return the size of an array.

Description: This function divides the size of the array passed to it by the size of its first elements to find the number of elements in the array.

Note, this function is implemented as a macro.

Language: C

Call:

(Int) = DitsNumber (array)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

(>) array (any array type) Specifies a fixed size array.

Include files: DitsTypes.h

External functions used: None

External values used: None

Prior requirements:

Support: Tony Farrell, AAO

C.102 DitsObey — Send an OBEY message to a task

Function: Send an OBEY message to a task

Description: An OBEY message with the specified action name is sent to the task on the given path. The specified argument is put in the message.

An OBEY message can only be sent if the action to which it is directed is not already active. The invoked action is made a subsidiary action of the current action. The subsidiary action's obey routine will be invoked.

If the OBEY fails, a "transaction failure" message will be received by the task with the transaction id returned by this routine. Otherwise a completion message will be received when the subsidiary action completes and the current action should reschedule to await this. Other messages (trigger messages, error messages and informational messages) may also be received and cause the action to reschedule, depending upon which of these the action is interested in (see DitsInterested and DitsNotInterested).

Note, this function is implemented using a macro to invoke DitsInitiateMessage.

Language: C

Call:

(Void) = DitsObey(path,action,argument,transid, status)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(>) path (DitsPathType) The path to the task to which the message is directed. See DitsGetPath.

(>) action (char *) The name of the action to obey.

(>) argument (SdsIdType) An argument to the action. This should be an Sds id. See DitsGetArgument and DitsPutArgument for more details on action arguments.

(<) transid (DitsTransIdType *) Transaction id for the transaction started. If an address of zero is supplied, then don't create a transaction id. In this case we will never be notified of transaction errors

(!) status (StatusType *) Modified status.

Include files: DitsInteraction.h

External functions used:

DitsInitiateMessage	Dits internal	Send a message to a task.

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or when the context if DITS_CTX_UFACE.

See Also: The Dits Specification Document, DitsInitiateMessage(3), DitsKick(3), DitsSetParam(3), DitsGetParam(3), DitsPathGet(3), DuiExecuteCmd(3), DulMessageW(3)

Support: Tony Farrell, AAO

C.103 DitsPathGet — Initiates the getting of or returns a path to a task.

Function: Initiates the getting of or returns a path to a task.

Description: A path is required for a Dits task to send messages to other Dits tasks. This routine will either return a path to the specified task or initiate actions to obtain the path.

If we can initiate the get path operation, then a transaction is started. The transaction id is returned in transid and the path is returned in path, although it will not be usable until the transaction completes.

The user action calling this routine should reschedule to await this message, which will have a reason code of `DITS_REA_PATHFOUND` or `DITS_REA_PATHFAILED`. The entry transaction id will be the transaction id as returned by this routine.

If the flag `DITS_M_PG_IMMED` is set, then this routine will return without setting up a transaction in two cases. The first occurs if we are already getting a path to the same task, in which case, status is returned set to `DITS__FINDINGPATH`. The second occurs if we already have a valid path to this task. In that case, we return with status ok and transid set to 0. The `DITS_M_PG_IMMED` flag causes this routine to emulate the behavious of versions of `DRAMA` prior to `V1`.0.

By specifing 0 for the transid, we can inquire about and existing path. In this case, the message sizes are ignored. We analyze the task name and try to find an existing path to the specified task. If we fail, status is set to `DITS__UNKNTASK` and path to 0. If we find an existing path, then we return it, but will set status to `DITS__INVPATH` if the path is not available for sending (probably waiting for a connection to be set up). In the later case, the only sensible thing to do with the path would be to lose it using DitsLosePath. The info argument is ignored during inquiries.

Please note that there is only ever one path between any two tasks and the buffer sizes are as set up for the first path to be set up between the two tasks. Subsequent calls to set up a path just returns the existing path, using the original buffer sizes. We consider this a flaw and believe the buffer sizes should be the maximum specified in any such calls, but we don't have a way of implementing this at this time. Note that if a given task has been sending connections to your task, then a connection has already been set up.

Language: C

Call:

(Void) = DitsPathGet (name, node, flags, info ,path, transid, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) name (char *) The null terminated name the task is thought to be registered by.

(>) node (char *) The node the task is running on. If not supplied then either the task is already known local (running on the local machine or some local task already has a path to it) or the node is specified as part of the name item in the normal internet format.

(>) flags (int) Selects various options to the PathGet operation. These are one of

DITS_M_FLOW_CONTROL	The connection should be flow controlled. See the Imp documentation for more details about the effect of flow control.
DITS_M_PG_IMMED	If set, then two cases which 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 normaly set up a transaction to complete when the path is available for sending. If this flag is set, we return immeidatly with no transaction set up but status ok. *transid will be zero.

(>) info (DitsPathInfoType *) Use this structure to supply various details for the GetPath operation. The following values should be set.

MessageBytes	The maximum number of bytes anticipated for any message to be sent.
MaxMessages	The maximum number of messages of size MaxBytes to be queued.
ReplyBytes	This is the maximum number of bytes anticipated in any messages sent back to the calling task.
MaxReplies	The maximum number of replies of size ReplyBytes anticipated.

(<) path (DitsPathType *) The path id is written here. Full details of the path may or may not be known immediately. If the task to which a path is required is already known then this path will be valid immediately and transid will be 0. If the task is remote and not yet known then messages must be sent to find it. When this happens, transid will contain a transaction id and the path will not become valid until an appropriate message is received. The user's action routine should reschedule to await invocation with a reason of DITS_REA_PATHFOUND or DITS_REA_PATHFAILED. Once created, a path is not deleted until the the current task exits, although it will become invalid if the target task disconnects.

(<) transid (DitsTransIdType *) If the path is known immediately and the DITS_M_PG_IMMED flag is set, then this will be set to zero. If not, it will contain the transaction id of the transaction required to find the path. If not this pointer is not supplied, then just do an inquiry.

(!) status (StatusType *) Modified status.

External functions used:

Dits___PathByTask	Dits internal	Find a path given the task name
Dits___PathCreate	Dits internal	Create a new path
Dits___PathSetName	Dits internal	Set the name of a path
Dits___PathSetTransId	Dits internal	Set the transaction id for path creation.
Dits___PathSetStatus	Dits internal	Set a path status
Dits___PathDelete	Dits internal	Delete a path
Dits___PathSetConnection	Dits internal	Set the connection of a path.
Dits___TransIdCreate	Dits internal	Create a new transaction id.
Dits___TransIdDelete	Dits internal	Delete a transaction id.
ImpConnect	Imp	Connect to a task
ImpNetLocate	Imp	Locate a remote task.

Include files: DitsInteraction.h

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or when the context if `DITS_CTX_UFACE`.

See Also: The Dits Specification Document, DitsGetPath(3), DitsAppInit(3), DitsInitiateMessage(3), DulGetPathW(3), DuiExecuteCmd(3), DitsPathGetInit(3).

Support: Tony Farrell, `AAO`

C.104 DitsPathToNode — Given a `DITS` Path, return the node name.

Function: Given a `DITS` Path, return the node name.

Description: This routine attempts to get the node associated with a given path name. If this fails, it will return the `IP` address as a string.

Language: C

Call:
(void) = DitsPathToNode (path, len, name, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) path (DitsPathType) The path to get the name of.

(>) len (int len) amount of space in name.

(>) name (char *) Where to write the name.

(!) status (StatusType *) Modified status.

Include File: DitsUtil.h

Prior requirements: DitsAppInit() must have been called.

See Also: The Dits Specification Document, DitsPathGet(3).

Support: Tony Farrell, AAO

C.105 DitsPeek — Peek at queue messages for any of interest.

Function: Peek at queue messages for any of interest.

Description: The routine allows an action to peek at the queue of incomming messages for any of interest. For example, a time consuming action may wish to check if the user has requested it abort by sending a KICK.

The user specifies a mask of flags indicating which events it is interested in.

DitsPeek will transparently handle some messages, if they do not require an action to be invoked and will not effect the current action. GET, MONITOR and most CONTROL messages are handled as are various other internal messages.

CONTROL messages which change the directory are not handled as this could effect the current action.

SET messages will be handled if the appropriate flag is set.

Language: C

Call:
(void) = DitsPeek (flags,namelen,name,arg,found,status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) flags (int) A mask of the following bits which determines what we are peeking for.

DITS_M_PEEK_OBEY	Look for incomming obeys
DITS_M_PEEK_KICK	Look for incomming kicks
DITS_M_PEEK_RESCH	Look for incomming reschedule messages (from a timeout or DitsSignal Call)
DITS_M_PEEK_SUBSID	Look for incomming messages related to a subsidiary action, a load request or a get path operation.
DITS_M_PEEK_THISACT	Look for messages, the associated name of which is the name of the invoking action.
DITS_M_PEEK_GETARG	If a message is found and it has an argument, return it.
DITS_M_PEEK_ALLOWSETPAR	If set, DitsPeek will transparently handle Set parameter messges in the context of this call.

If none of the first 4 flags is supplied, all are assumed.

(>) namelen (int) The number of bytes in the name argument.

(<) name (char *) The name associated with any found message is written here.

(<) arg (SdsIdType *) If the flag DITS_M_PEEK_GETARG is set, the Sds id of any argument attached to the message is returned here. IF none exists, or the flag is not set, it is set to zero. The caller should do an SdsDelete and SdsFreeId on the argument when it is finished with it.

(<) found (int) Set to the flag (as per first 4 above) which indicates the type of message found. IF zero, nothing was found.

(!) status (StatusType *) Modified status.

Include Files: DitsPeek.h

External functions used:

External values used: DitsTask

Prior requirements: Can only be called from a Dits application routine or a user interface response routine

See Also: The Dits Specification Document, DitsMainLoop(3), DitsInitiateMessage(3).

Support: Tony Farrell, AAO

C.106 DitsPrintReason — Converts a reason code and status into a string and outputs an

Function: Converts a reason code and status into a string and outputs an appropriate message

Description: This is a useful debugging routine. The reason code is converted to a string and output using ErsOut or MsgOut. If the code is a failure code, then the status is also output.

Language: C

Call:
(Void) = DitsPrintReason (reason, reasonstat, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) reason (DitsReasonType) The reason code

(>) reasonstat (StatusType) The status associate with the reason

(!) status (StatusType *) Modified status.

Include files: DitsFix.h

External functions used:

ErsOut	Ers	Output an error message.
MsgOut	Dits	Output a message.
DitsFmtReason	Dits	Format a an entry message reason.
DitsErrorText	Dits	Return the text assocaited with an error.

External values used: none

Prior requirements: none

See Also: The Dits Specification Document, DuiLogEntry(3), DitsFmtReason(3), DitsGetEntInfo(3).

Support: Tony Farrell, `AAO`

C.107 DitsPutActData — Stores an item of data assocaited with the current action.

Function: Stores an item of data assocaited with the current action.

Description: Saves an item which can be retreived with a later call to DitsGetActData when called from an action handler for the same action.

This call allows data to be stored between reschedules of an action and even between different invocations of the same action. This is of particular interest on machines with a common address space accross all tasks, e.g. VxWorks.

Language: C

Call:
(Void *) = DitsPutActData (data, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) data (void `*`) The data to store.

(`!`) status (StatusType `*`) Modified status.

Include files: DitsFix.h

Return value: The previous value of Act data.

External functions used: None

External values used: DitsTask - Details of the current task

See Also: The Dits Specification Document, DitsGetActData(3), DitsPutUserData(3), DitsPutTransData(3), DitsPutPathData(3).

Prior requirements: Can only be called from a Dits application routine

Support: Tony Farrell, `AAO`

C.108 DitsPutActDescr — Set the description of the specified action.

Function: Set the description of the specified action.

Description: Set the description of the specified action. This string is normally set when the action is created. It is meant to work as a simple help string for an action. This function allows string description to be changed.

Language: C

Call:
(Void) = DitsPutThisActDescr (name, descr, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) name (const char `*`) The name of the action to change the description for.

(`>`) descr (const char `*`) If not null. Specifies a description for the specified action. Only the first `DITS_ACT_DESCR_LEN` (79) characters are used. Can be fetched using DitsGetActDescr(3) and is output by control messages such as `LISTACTALL` and `LISTACT`.

(`!`) status (StatusType `*`) Modified status.

Include files: DitsFix.h

External functions used:

strncpy	`CRTL`	Copy one string to another.

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine

See Also: The Dits Specification Document, DitsPutActions(3), DitsPutAction(3), DitsGetActDescr(3), DitsPutActDescr().

Support: Tony Farrell, `AAO`

C.109 DitsPutActEndRoutine — Put a routine to be invoked when the action completes.

Function: Put a routine to be invoked when the action completes.

Description: This function inserts a function which will be invoked whenever the invoking action completes. This allows the action to catch cases where it completing regardless of why it is completing.

The action end routine is cleared when the action starts, so must be inserted afresh on each invocation of the action.

The action end routine is called after any bulk data has tidyed up but before any remaining `ERS` messages are flushed (so they can be annuled or added to) and before the action completion message is returned.

The completion of the action would have already been logged if a logging system is active.

Language: C

Call:
(void) = DitsPutActEndRoutine(routine,client_data, old_routine,old_c_data,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(>) routine (DitsActEndRoutineType)** The new action end routine. See Dits documentation for details.
- **(>) client_data (Void *)** Passed directly to the routine as its client_data argument.
- **(!) old_routine (DitsActEndRoutineType *)** If non-null, return the previous action end routine address here. This address may be null.
- **(!) old_c_data (Void **)** If non-null, return the previous action end routine client data here.
- **(!) status (StatusType *)** Modified status.

Function Prototypes Used: typedef `DVOID` (*DitsActEndRoutineType)(`DVOIDP` client_data, int taskExit, StatusType exitStatus, StatusType *status);

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) client_data (void *) The value passed to DitsPutActEndRoutine().

(>) taskExit (int) Set to true (1) if the task is exiting.

(>) exitStatus (StatusType) The action exit status.

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used: None

External values used: DitsTask

Prior requirements: Can only be called from a Dits application routine.

See Also: The Dits Specification Document.

Support: Tony Farrell, `AAO`

C.110 DitsPutAction — Register an action handler.

Function: Register an action handler.

Description: Saves the details of and action names and handler. Space is allocated to store the details. Multiple calls may be made to this routine. Later calls can redefine action names defined in earlier calls. Note that when this happens, the space used by the earlier action definition is `NOT` recovered.

There is an older call which does a simlar job, DitsPutActionHandlers(3). That routine does not allow spawnability. There is a version which allows multiple actions to be added in one go, which is more efficient, see DitsPutActions(3).

Language: C

Call:

(Void) = DitsPutAction (name, obey, kick, code, spwanable, spwanCheckRoutine, spawnCheck-Data, cleanupRoutine ,actDescr, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) name (const char *) The name the action. Will be truncated to `DITS_C_NAMELEN` characters

(>) obey (DitsActionRoutineType) The routine to be invoked when a obey message is received for this action.

(>) kick (DitsActionRoutineType) The routine to be invoked when a Kick message is received for this action. If null, then by default Kicks are reject, but this can be changed by the obey handler invoking DitsPutKickHandler3)

(>) code (long int) The code assocaited with this action, which can be fetched by Dits-GetCode() when the action is invoked.

(>) **spawnable (int)** If true (non-zero) indicates the action is spawable, which allows multiple invocations of actions of the same name to be outstanding at the same time. (Note, spawnable actions must be signalled and kicked in a more complex way then normal. See the dits specification document).

(>) **spawnCheckRoutine (DitsSpawnCheckRoutineType)** If not null, then before spawning a new version of this action, this routine is called. If it returns with status ok, then the spawn is allowed. Otherwise, it is disallowed. This allows a program to limit the number of invocations of a spawnable action.

`DITS_M_CLEANUP` Specifies that the info structure contains the address of a routine to be invoked when the action is destroyed. Note that `DITS_M_CLEANUP` being set also causes `DITS_M_ACT_INFO` to be set.

(>) **spawnCheckData (void *)** The client data item for the spawnCheckRoutine.

(>) **cleanupRoutine (DitsActionCleanupRoutineType)** If not null, is the address of a routine which is invoked when the action structure is destroyed. This occurs when the program is shutdown by calling DitsStop. It also occurs at the end of each action when the action is spawnable.

(>) **actDescr (const char)** If not null. Specifies a description for the action. Only the first `DITS_ACT_DESCR_LEN` (79) characters are used. Can be fetched using DitsGetActDescr(3) and is output by control messages such as `LISTACTALL` and `LISTACT`.

(!) **status (StatusType *)** Modified status.

Function Prototypes Used: typedef `DVOID` (*DitsActionRoutineType)(StatusType *status);

typedef `DVOID` (*DitsSpawnCheckRoutineType)(`DVOIDP` client_data, StatusType *status);

typedef `DVOID` (*DitsActionCleanupRoutineType)(long int code, StatusType *status);

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) **client_data (void *)** Will contain the data item associated with the spawn check routine (which may be the spawnData value in a structure of type DitsActInfoType, if the `DITS_M_ACT_INFO` when DitsPutActions() was invoked).

(>) **code (long)** The code assoicated with the action.

(!) **status (StatusType *)** Modified status. The action obey/kick handler function should set status bad when it fails - this status will be returned to the task which sent the obey/kick message.

Include files: DitsSys.h

External functions used:

malloc	Crtl	Allocate memory.
free	Crtl	Free memory.
Dits___NameSet	Dits internal	Set a Dits name string from a C string.

External values used: DitsTask - Details of the current task

Prior requirements: Can be called any number of times after calling DitsInit(3)/DitsAppInit(3) but before invoking DitsMainLoop(3)/DitsMsgReceive(3)/DitsAltInLoop(3) or any higher level routine which invokeds the `DRAMA` event loop.

See Also: The Dits Specification Document, DitsPutActionHandlers(3), DitsAppInit(3).

Support: Tony Farrell, `AAO`

C.111 DitsPutActionHandlers — Register action handlers, Obsolete, See DitsPutActions(3).

Function: Register action handlers, Obsolete, See DitsPutActions(3).

Description: Saves the details of action names and handlers. Space is allocated to store the details. Multiple calls may be made to this routine. Later calls can redefine action names defined in earlier calls. Note that when this happens, the space used by the earlier action definition is `NOT` recovered.

Also see DitsPutActions(3).

Language: C

Depreciated: See DitsPutActions(3) and DitsPutAction(3).

Call:
(Void) = DitsPutActionHandlers (size,map,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) size (Long int) The number of elements in map

(>) map (DitsActionMapType`[]`) Details of actions. This is an array with each element containing the following elements in this order.

obey	(DitsActionHandler) The routine to be invoked when a obey message is received for this action.
kick	(DitsActionHandler) The routine to be invoked when a Kick message is received for this action.
code	(long int) The code assocaited with this action, which can be fetched by DitsGetCode() when the action is invoked.
name	(char `*`) The name of the action. This should be no more then `DITS_C_NAMELEN` characters (20 characters) long.

(!) status (StatusType `*`) Modified status.

Include files: DitsSys.h

External functions used:

malloc	Crtl	Allocate memory.
free	Crtl	Free memory. DitsPutActions

External values used: none

Prior requirements: Can be called any number of times after calling DitsInit(3)/DitsAppInit(3) but before invoking DitsMainLoop(3)/DitsMsgReceive(3)/DitsAltInLoop(3) or any higher level routine which invokes the `DRAMA` event loop.

See Also: The Dits Specification Document, DitsPutActions(3), DitsAppInit(3).

Support: Tony Farrell, `AAO`

C.112 DitsPutArgument — Set the argument to return if the action completes.

Function: Set the argument to return if the action completes.

Description: Called by an Action routine to set the argument structure to be returned if this action completes.

It saves the argument for use when the action returns. This routine only has an effect if an appropriate request is put with DitsPutRequest. The appropriate requests are

`DITS_REQ_END`	Action to complete immediately
`DITS_REQ_EXIT`	Action to complete immediately and task to exit.

The argument is encoded in Sds, hence this routine simply does an SdsCopy of the argument or saves its id depending upon the flag.

There are no restrictions on the argument, any value which can encoded in Sds may be supplied as an argument, although there may be limitations to the size of messages which can be sent. See DitsAppInit(3) and DitsPathGet(3) for details of such limitations.

Normally, arguments are encoded and decoded using the `ARG` package (supplied as part of `SDS`), but this is not required.

Note, this function is implemented as a macro in C.

Language: C

Call:

(Void) = DitsPutArgument (argument, flag, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) argument (SdsIdType) The argument to return. This should be the Sds Id of the argument.

(>) flag (DitsArgFlagType) Possible values are

`DITS_ARG_COPY`	A copy is made of the argument using SdsCopy and this copy is deleted after the action completes.
`DITS_ARG_DELETE`	No copy is made, the original argument is deleted after the action completes using SdsDelete, followed by SdsFreeId(). You should not delete this item yourself.
`DITS_ARG_READFREE`	No copy is made, the original argument is read-free-ed after the action completes using SdsReadFree(), followed by SdsFreeId().
`DITS_ARG_FREEID`	No copy is made. the original argument is free-ed using SdsFreeId() after the action completes.
`DITS_ARG_NODELETE`	The argument is not copied and not deleted and not read-free-ed. You should not delete this item yourself until after this reschedule of this action exit.

(!) status (StatusType *) Modified status.

Include files: DitsFix.h

External functions used:

SdsCopy	`SDS`	Makes a copy of the argument.

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine

See Also: The Dits Specification Document, DitsGetArgument(3), Sds manual.

Support: Tony Farrell, `AAO`

C.113 DitsPutCode — Change the value return by DitsGetCode.

Function: Change the value return by DitsGetCode.

Description: Change the value returned by DitsGetCode for the current action.

Language: C

Call:
(void) = DitsPutCode (code,status)

Include files: DitsFix.h

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or a user interface response routine.

See Also: The Dits Specification Document, DitsGetCode(3), DitsPutActions(3), DitsPutActionHandlers(3).

Support: Tony Farrell, `AAO`

C.114 DitsPutConnectHandler — Put a routine to be invoked when another task attempts to connect.

Function: Put a routine to be invoked when another task attempts to connect.

Description: This function inserts a function which will be invoked whenever another task is attempting to connect to the task. It allows the current task to reject the connection or modify the message buffer sizes. Note that this routine is not informed about connections initiated by this task.

The previous handler's address and client data are returned allowing you to link operations.

If the handler routine returns true, then connection is allowed. Otherwise, the connection is not allowed.

The connect routine is invoked in `UFACE` context. To send `DRAMA` messages from this routine, you should first invoke DitsUfaceCtxEnable() to specify an appropriate handler for reponses to your message. Alternatively, if you want to communicate with an action in your task which is already running, you can use one of the DitsSignal() series of routines or use a shared variable that will be examined the next time the action is rescheduled. (You can communicate with a running action in your task by sending it a Kick message, but DitsSignal() is the prefered approach, as it is a simpler interface since and you don't need the call to DitsUfaceCtxEnable() or a path to your task).

Language: C

Call:
(void) = DitsPutConnectHandler(routine,client_data, old_routine,old_c_data,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) routine (DitsConnectRoutineType) The new connect routine.

(>) client_data (Void *) Passed directly to the routine as its client_data argument.

(!) old_routine (DitsConnectRoutineType *) If non-null, return the previous connect handler address here. This address may be null.

(!) old_c_data (Void **) If non-null, return the previous connect handlers client data here.

(!) status (StatusType *) Modified status.

Function Prototypes Used: typedef int (*DitsConnectRoutineType)(`DVOIDP` client_data, `DCONSTV` char * taskname, DitsConnectInfoType *ConnectInfo, int *flags, StatusType * status);

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) client_data (void *) The value passed to DitsPutConnectHandler().

(>) taskname (const char *) The name of the connecting task.

(!) ConnectInfo (DitsConnectInfoType *) The items MessageBytes and MaxMessages can be set to specify the sizes of the buffers for the reply connection. If not set, then the values set by the requesting task are used (I'm afraid you can't work out the values requested by the requesting task as yet).

(!) flags (int *) Allows flags to be specified for this connection. The possible values are

`DITS_M_FLOW_CONTROL`	Specifies that a flow-controlled connection is to be established. Note that if a connect handler is `NOT` used, then the status of this flag is determined by the `DITS_M_NO_FC_AC` flag to DitsAppInit().

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used: None

External values used: DitsTask

Prior requirements: DitsAppInit should have been called.

See Also: The Dits Specification Document, DitsPutDisConnectHandler(3), DitsPathGet(3).

Support: Tony Farrell, `AAO`

C.115 DitsPutDefCode — Sets a new code to be associated with a named action.

Function: Sets a new code to be associated with a named action.

Description: Sets a code routine to be assocaited when the specified action is invoked. This will now impact a running action, its code won't change until the next time the action is started.

Language: C

Call:
(long int) = DitsPutDefCode (name, code, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) name (char *) Name of the action who's obey handle is to be changed.

(>) code (long int) The new action code.

(!) status (StatusType *) Modified status.

Include files: DitsFix.h

Return value: The previous code value for the action..

External functions used:

Dits___ActptrByName	Dits internal	Return the action pointer associated with an action name.

External values used: DitsTask - Details of the current task

Prior requirements: DitsAppInit must have been called.

See Also: The Dits Specification Document, DitsPutDefKickHandler(3), DitsPutObeyHandler(3), DitsPutActions(3), DitsPutCode(3), DitsGetCode(3).

Support: Tony Farrell, `AAO`

C.116 DitsPutDefKickHandler — Sets a new kick handler routine for an action.

Function: Sets a new kick handler routine for an action.

Description: Sets a new routine to be called when the action is invoked. Does not change the current kick handler for an active action, just the routine called after it is next OBEYed.

Language: C

Call:
(DitsActionRoutineType) = DitsPutDefKickHandler (name, routine, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) name (char *) Name of the action who's kick handle is to be changed.

(>) routine (DitsActionRoutineType) The handler routine. See the Dits documentation for details.

(!) status (StatusType *) Modified status.

Function Prototypes Used: typedef void (*DitsActionRoutineType)(StatusType *status);

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(!) status (StatusType *) Modified status. The action kick handler function should set status bad when it fails - this status will be returned to the task which sent the kick message.

Include files: DitsFix.h

Return value: The previous default kick handler.

External functions used:

Dits___ActptrByName	Dits internal	Return the action pointer associated with an action name.

External values used: DitsTask - Details of the current task

Prior requirements: DitsAppInit must have been called.

See Also: The Dits Specification Document, DitsPutDefObeyHandler(3), DitsPutKickHandler(3), DitsPutActions(3).

Support: Tony Farrell, `AAO`

C.117 DitsPutDefObeyHandler — Sets a new obey handler routine for an action.

Function: Sets a new obey handler routine for an action.

Description: Sets a new routine to be called when the action is invoked. Does not change the current obey handler for an active action, just the routine called after it is next OBEYed.

Language: C

Call:
(DitsActionRoutineType) = DitsPutDefObeyHandler (name, routine, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) name (char *) Name of the action who's obey handle is to be changed.

(>) routine (DitsActionRoutineType) The handler routine. See the Dits documentation for details.

(!) status (StatusType *) Modified status.

Function Prototypes Used: typedef void (*DitsActionRoutineType)(StatusType *status);

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(!) status (StatusType *) Modified status. The action obey handler function should set status bad when it fails - this status will be returned to the task which sent the obey message.

Include files: DitsFix.h

Return value: The previous default obey handler.

External functions used:

Dits___ActptrByName	Dits internal	Return the action pointer associated with an action name.

External values used: DitsTask - Details of the current task

Prior requirements: DitsAppInit must have been called.

See Also: The Dits Specification Document, DitsPutDefKickHandler(3), DitsPutObeyHandler(3), DitsPutActions(3).

Support: Tony Farrell, `AAO`

C.118 DitsPutDefaultHandler — Sets a new handler for `DEFAULT` control messages.

Function: Sets a new handler for `DEFAULT` control messages.

Description: Sets the handler for control messages with the name `DEFAULT`, which sets or returns the default directory. At initialisation, the routine DitsDefault is used.

The previous handler's address and client data are returned allowing you to link operations.

Language: C

Call:

(void) = DitsPutDefaultHandler(routine,client_data, old_routine,old_c_data,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(`>`) routine (DitsDefaultRoutineType)** The new default routine. See Dits documentation for details.
- **(`>`) client_data (Void *)** Passed directly to the routine as its client_data argument.
- **(`!`) old_routine (DitsDefaultRoutineType *)** If non-null, return the previous default handler address here. This address may be null.
- **(`!`) old_c_data (Void **)** If non-null, return the previous connect handlers client data here.
- **(`!`) status (StatusType *)** Modified status.

Function Prototypes Used: typedef `DVOID` (*DitsDefaultRoutineType)(`DVOIDP` client_data, `DCONSTV` char * newdir, int resultlen, char * result, StatusType * status);

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(`>`) client_data (void *)** The value passed to DitsPutDefaultHandler().
- **(`>`) newdir (const char *)** The new directory
- **(`>`) resultlen (int)** Space in result
- **(`<`) result (char *)** The resultant directory.
- **(`!`) status (StatusType *)** Modified status.

Include files: DitsFix.h

External functions used: None

External values used: DitsTask

Prior requirements: DitsAppInit should have been called.

See Also: The Dits Specification Document, DitsDefault(3).

Support: Tony Farrell, `AAO`

C.119 DitsPutDelay — Set Action delay or timeout.

Function: Set Action delay or timeout.

Description: Called by an Action routine to set the Delay before the next reschedule of this action.

Simply saves the delay for use when the action returns. Note that this routine only has an effect if an appropriate request was put with DitsPutRequest. The appropriate requests are

DITS_REQ_WAIT	Standard wait
DITS_REQ_SLEEP	Timeout
DITS_REQ_MESSAGE	Timeout

The first is the standard for of polling. For the the later two, the delay acts as a timeout.

The actual length of the delay is a function of the underlying hardware - A minimum non-zero delay will normally apply as will a minimum resolution.

Note, this function is implemented as a macro in C.

Language: C

Call:
(Void) = DitsPutDelay (delay, status)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(>) delay (DitsDeltaTimeType *) The delay before rescheduling. See DitsDeltaTime for details on how to create this value.

(!) status (StatusType *) Modified status.

Include files: DitsFix.h

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine

See Also: The Dits Specification Document, DitsDeltaType(3), GitPutDelay(3), DitsPutDelayPar(3).

Support: Tony Farrell, AAO

C.120 DitsPutDisConnectHandler — Put a routine to be invoked when another task disconnects from

Function: Put a routine to be invoked when another task disconnects from us.

Description: This function inserts a function which will be invoked whenever another task is diconnects from this task. connections initiated by this task.

The previous handler's address and client data are returned allowing you to link operations.

Note that the disconnect handler is invoked quite early in the process of handling the disconnection, before action/UFACE code, which will be scheduled as a result, is invoked.

The disconnect handler is invoked in UFACE context. To send DRAMA messages from this routine, you should first invoke DitsUfaceCtxEnable() to specify an appropriate handler

for reponses to your message. Alternatively, if you want to communicate with an action in your task which is already running, you can use one of the DitsSignal() series of routines or use a shared variable that will be examined the next ‘ time the action is rescheduled. (You can communicate with a running action in your task by sending it a Kick message, but DitsSignal() is the prefered approach, as it is a simpler interface since and you don't need the call to DitsUfaceCtxEnable() or a path to your task).

The disconnect handler can also use DitsPutRequest(`DITS_REQ_EXIT`,status) to indicate that this task should now exit. In that case, the status returned by the disconnect handler will be the task exit status.

Language: C

Call:
(void) = DitsPutDisConnectHandler(routine,client_data, old_routine,old_c_data,status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

(>) routine (DitsDisConnectRoutineType) The new connect routine.

(>) client_data (Void *) Passed directly to the routine as its client_data argument.

(!) old_routine (DitsDisConnectRoutineType *) If non-null, return the previous connect handler address here. This address may be null.

(!) old_c_data (Void **) If non-null, return the previous connect handlers client data here.

(!) status (StatusType *) Modified status.

Function Prototypes Used: typedef `DVOID` (*DitsDisConnectRoutineType)(`DVOIDP` client_data, `DCONSTV` char * taskname, DitsPathType path, StatusType * status);

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

(>) client_data (void *) The value passed to DitsPutDisConnectHandler().

(>) taskname (const char *) The name of the disconnecting task.

(>) path (DitsPathType) The path to the disconnecting task.

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used: None

External values used: DitsTask

Prior requirements: DitsAppInit should have been called.

See Also: The Dits Specification Document, DitsPutConnectHandler(3), DitsPathGet(3).

Support: Tony Farrell, `AAO`

C.121 DitsPutEventWaitHandler — Put a routine to be invoked when waiting for events.

Function: Put a routine to be invoked when waiting for events.

Description: This routine puts a function which is invoked by DitsActionWait or DitsUfaceWait when they want to wait for events. This routine should return when a `DRAMA` event occurs. It allows other events (say X windows events) to be handled while while DitsActionWait or DitsUfaceWait are active.

Note, this function is implemented as a macro.

Language: C

Call:
(Void) = DitsPutEventWaitHandler (routine,client_data,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) routine (DitsWaitRoutineType) The wait routine.

(>) client_data (Void *) Passed directly to the routine as its client_data argument.

(!) status (StatusType *) Modified status.

Function Prototypes Used: typedef `DVOID` (*DitsWaitRoutineType)(`DVOIDP` client_data, StatusType *status);

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) client_data (void *) The value passed to DitsPutEventWaitHandler().

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used: None

External values used: DitsTask

Prior requirements: DitsAppInit should have been called.

See Also: The Dits Specification Document, DitsActionWait(3), DitsUfaceWait(3).

Support: Tony Farrell, `AAO`

C.122 DitsPutForwardMonSetupHandler — Put a routine to be invoked when another a forward monitor path is being

Function: Put a routine to be invoked when another a forward monitor path is being setup.

Description: This function inserts a function which will be invoked whenever another task is attempting forward monitor parameters from this task to another task and the path is being established. It allows the current task to modify the space to be allowed for the messages to that task.

Language: C

Call:
(void) = DitsPutForwardMonSetupHandler(routine,client_data, old_routine,old_c_data,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) routine (DitsForwardMonSetupRoutineType) The new routine.

(>) client_data (Void *) Passed directly to the routine as its client_data argument.

(!) old_routine (DitsForwardMonSetupRoutineType *) If non-null, return the previous connect handler address here. This address may be null.

(!) old_c_data (Void **) If non-null, return the previous connect handlers client data here.

(!) status (StatusType *) Modified status.

Function Prototypes Used: typedef `DVOUD` (*DitsForwardMonSetupRoutineType)(`DVOIDP` client_data, `DCONSTV` char * taskname, `DCONSTV` char * action, `DCONSTV` SdsIdType parameters, int * bufferSize, StatusType * status);

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) client_data (void *) The value passed to DitsPutForwardMonSetupHandler().

(>) taskname (const char *) The name of the task to forward monitor is directed to.

(>) action (const char *) The name of the action the forward monitor is directed to.

(>) parameters (SdsIdType argin) Argument to the message. The first item is the target task name, the second the target action and any subsequent items are the names of parameters to be monitored.

(!) bufferSize (int *) On input, the number of bytes being suggested by the monitor function to allow space for writing messages into the target task (normally the size of the largest parameters + 1000. To override this value, set the new value here.

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used: None

External values used: DitsTask

Prior requirements: DitsAppInit should have been called.

See Also: The Dits Specification Document, DitsPutConnectHandler(3), DitsInitiateMessage(3).

Support: Tony Farrell, AAO

C.123 DitsPutKickHandler — Sets a new handler routine for the next kick.

Function: Sets a new handler routine for the next kick.

Description: Called by an Action routine to set the routine to be called on the next kick of this action.

Simply saves the routine.

Language: C

Call:
(DitsActionRoutineType) = DitsPutKickHandler (routine, status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) routine (DitsActionRoutineType) The handler routine. See the Dits documentation for details. If zero, then kick messages to this action will be rejected.

(!) status (StatusType *) Modified status.

Function Prototypes Used: typedef void (*DitsActionRoutineType)(StatusType *status);

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(!) status (StatusType *) Modified status. The action kick handler function should set status bad when it fails - this status will be returned to the task which sent the kick message.

Include files: DitsFix.h

Return value: The previous kick handler.

External functions used: none

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine

See Also: The Dits Specification Document, DitsPutActions(3), DitsPutObeyHandler(3), DitsPutDefKickHandler(3).

Support: Tony Farrell, AAO

C.124 DitsPutObeyHandler — Sets a new handler routine for the next reschedule.

Function: Sets a new handler routine for the next reschedule.

Description: Called by an Action routine to set the routine to be called on the next reschedule of this action.

Simply saves the routine.

The old name DitsPutHandler should be replaced by DitsPutObeyHandler in new code.

Language: C

Call:
(DitsActionRoutineType) = DitsPutObeyHandler (routine, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) routine (DitsActionRoutineType) The handler routine. See the Dits documentation for details.

(!) status (StatusType *) Modified status.

Function Prototypes Used: typedef void (*DitsActionRoutineType)(StatusType *status);

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(!) status (StatusType *) Modified status. The action obey handler function should set status bad when it fails - this status will be returned to the task which sent the obey message.

Include files: DitsFix.h

External functions used: None

External values used: DitsTask - Details of the current task

Return value: The previous obey handler.

Prior requirements: Can only be called from a Dits application routine

See Also: The Dits Specification Document, DitsPutActions(3), DitsPutKickHandler(3), DitsPutDefObeyHandler(3).

Support: Tony Farrell, `AAO`

C.125 DitsPutOrphanHandler — Set the routine to be invoked for orphan routines.

Function: Set the routine to be invoked for orphan routines.

Description: The specified routine will be invoked, in uface context, when an orphan transaction completes.

Language: C

Call:
(DitsActionRoutineType) = DitsPutOrphanHandler (routine,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) routine (DitsActionRoutineType) The handler routine.

(!) status (StatusType *) Modified status.

Function Prototypes Used: typedef `DVOID` (*DitsActionRoutineType)(StatusType *status);

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(!) status (StatusType *) Modified status. The handler function should set status bad when it fails - this status will be returned to the task which sent the obey/kick message.

Include files: DitsOrphan.h

External values used: DitsTask

Return value: The previous orphan handler.

Prior requirements: DitsAppInit must have been called

See Also: The Dits Specification Document, DitsCheckTransactions(3), DitsIsOrphan(3), DitsTakeOrphans(3), DitsForget(3).

Support: Tony Farrell, `AAO`

C.126 DitsPutParSys — Enable, Disable or change the parameter system used by Dits, Obsolete, See DitsAppParamSys(3).

Function: Enable, Disable or change the parameter system used by Dits, Obsolete, See DitsAppParamSys(3).

Description: The specified routines are use by the Dits fix part to respond to Dits Get/Set messages. Disable a parameter system by specifing `NULL` as the routine addresses.

Obsolute routine, see DitsAppParamSys(3).

Language: C

Call:
(void) = DitsPutParSys(getroutine, setroutine, parid, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) getroutine (DitsGetRoutineType) A routine to call to get parameter values

(>) setroutine (DitsSetRoutineType) A routine to call to set parameter values

(>) parid (long int) The first argument to the Set and Get parameter routines

(!) status (StatusType *) Modified status.

Include files: DitsParam.h

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: DitsAppInit must have been called.

See Also: The Dits Specification Document, DitsAppParamSys(3), DitsPutParSys(3), DitsAppInit(3), SdpCreate(3).

Support: Tony Farrell, `AAO`

C.127 DitsPutParamMon — Enable the parameter monitoring system Obsolete, See DitsAppParamSys(3).

Function: Enable the parameter monitoring system Obsolete, See DitsAppParamSys(3).

Description: Normally only invoked by a parameter systems to enable the parameter monitor system.

Obsolute routine, see DitsAppParamSys(3).

Language: C

Call:
(void) = DitsPutParamMon(MsgRoutine, DisconRoutine, TidyRoutine, CheckRoutine, SizeRoutine, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) MsgRoutine (DitsMonitorMsgRoutineType) Routine to be called when a monitor message is received. Normally DitsMonitorMsg.

(>) DisconRoutine (DitsMonitorDisconRoutineType) Routine to be called when a task has disconnected. Normally DitsMonitorDisconnect.

(>) **TidyRoutine (DitsMonitorTidyRoutineType)** Routine to be called when the task shuts down (from DitsStop). Normally DitsMonitorTidy.

(>) **CheckRoutine (DitsMonitorCheckRoutineType)** Optional routine which is called to check a parameter for which monitoring is requested is valid. Should return an invalid status if the name is not valid.

(>) **SizeRoutine (DitsMonitorSizeRoutineType)** Routine is is called to return the size which will be needed to send a message containing the named parameter.

(>) **GetRoutine (DitsMonitorGetRoutineType)** Routine is is called to return the value of the named parameter (by Sds id). This is only used on rare occasions.

(!) **status (StatusType *)** Modified status.

Include files: DitsParam.h

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: DitsAppInit must have been called. Should only be called once, when the parameter system is initialised.

See Also: The Dits Specification Document, DitsAppParamSys(3), DitsPutParSys(3), DitsAppInit(3), SdpCreate(3).

Support: Tony Farrell, AAO

C.128 DitsPutPathData — Associate an item with a path

Function: Associate an item with a path

Description: This call associates the specified data item with the specified path, which must have been returned by another DitsPathGet/DitsGetPath call. The item can be retrived by DitsGetPathData().

Note, this function is implemented as a macro.

Language: C

Call:

(Void) = DitsPutPathData(data,path,status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) **data (void *)** The data item to associated with the path

(>) **path (DitsPathType)** The path as returned by another Dits call.

(!) **status (StatusType *)** Modified status.

Include files: DitsInteraction.h

External functions used: None

External values used: None

Prior requirements: None.

See Also: The Dits Specification Document, DitsGetPathData(3), DitsPutUserData(3), DitsPutTransData(3), DitsPutActData(3).

Support: Tony Farrell, `AAO`

C.129 DitsPutRegistrationHandler — Put a routine to be invoked when another task registers with `IMP`.

Function: Put a routine to be invoked when another task registers with `IMP`.

Description: This function inserts a function which will be invoked whenever another task registers with `IMP`. This will only occur if this this has specified the `DITS_M_REGISTRAR` flag when it invoked DitsAppInit(). If you call this routine without having specified `DITS_M_REGISTRAR`, then the error code `DITS__REGISTRAR` will be returned.

The previous handler's address and client data are returned allowing you to link operations.

The registration handler is invoked in `UFACE` context. To send `DRAMA` messages from this routine, you should first invoke DitsUfaceCtxEnable() to specify an appropriate handler for reponses to your message. Alternatively, if you want to communicate with an action in your task which is already running, you can use one of the DitsSignal() series of routines or use a shared variable that will be examined the next time the action is rescheduled. (You can communicate with a running action in your task by sending it a Kick message, but DitsSignal() is the prefered approach, as it is a simpler interface since and you don't need the call to DitsUfaceCtxEnable() or a path to your task).

For this to work with tasks registering on remote machines, the `IMP` Startup file on that machine must specify that such messages are to be forwarded to this machine. For example, lines such at the following:

forward registrations to machine1 machine2 machine3

There may be mulitple such lines and as many machines as you want on each line.

Note that only tasks which are built against `IMP` Version 3.43 or later will send registration messages.

Language: C

Call:

(void) = DitsPutRegistrationHandler(routine,client_data, old_routine,old_c_data,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) routine (DitsRegistrationRoutineType) The new registration routine.

(>) **client_data (void *)** Passed directly to the routine as its client_data argument.

(!) **old_routine (DitsRegistrationRoutineType *)** If non-null, return the previous handler address here. This address may be null.

(!) **old_c_data (void **)** If non-null, return the previous connect handlers client data here.

(!) **status (StatusType *)** Modified status.

Function Prototypes Used: typedef int (*DitsRegistrationRoutineType)(`DVOIDP` client_data, `DCONSTV` char * taskName, `DCONSTV` char * nodeName, StatusType * status);

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) **client_data (void *)** The value passed to DitsPutRegistrationHandler().

(>) **taskName (const char *)** The name of the registering task.

(>) **nodeName (const char *)** The node the task registered on.

(!) **status (StatusType *)** Modified status.

Include files: DitsSys.h

External functions used: None

External values used: DitsTask

Prior requirements: DitsAppInit should have been called with the `DITS_M_REGISTRAR` flag set.

See Also: The Dits Specification Document, DitsAppInit(3), DitsPutConnectHandler(3), DitsPathGet(3).

Support: Tony Farrell, `AAO`

C.130 DitsPutRequest — Request to the fixed part for when the Action returns.

Function: Request to the fixed part for when the Action returns.

Description: Called by an Action routine to indicate to the fixed part what it should do when the action returns

Simply saves the request for action when the Action returns. Any previous request is overwritten without warning.

The default request is `DITS_REQ_END`, except for kick routines where the default is to reject (status `!= STATUS__OK`) or ignore the kick.

Note, this function is implemented as a macro in C.

Language: C

Call:
(Void) = DitsPutRequest (request, status)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(>) request (DitsReqType) The request, one of

DITS_REQ_END	The action is to complete immediately.
DITS_REQ_STAGE	The action is to reschedule immediately.
DITS_REQ_WAIT	The action is to reschedule on expiry. of a timer.
DITS_REQ_SLEEP	The action will be put too sleep. It can be working up by a kick message or a DitsSignal() call.
DITS_REQ_MESSAGE	The action will be rescheduled on reception of a message from a subsidiary action.
DITS_REQ_EXIT	The action will complete and the task should exit.

Only DITS_REQ_EXIT has any effect when this routine is called from a user interface response routine.

(!) status (StatusType *) Modified status.

Include files: DitsFix.h

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or a user interface response routine.

See Also: The Dits Specification Document, DitsPutActions(3), DitsInitiateMessage(3), DitsPutDelay(3), DitsSignalByName(3), DitsSignalByAction(3), DitsPutObeyHandler(3), DitsPutKickHandler(3).

Support: Tony Farrell, AAO

C.131 DitsPutThisActDescr — Set the description of the current action.

Function: Set the description of the current action.

Description: Set the description of the current action. This string is normally set when the action is created. It is meant to work as a simple help string for an action. This function allows string description to be changed.

Language: C

Call:
(Void) = DitsPutThisActDescr (descr, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) descr (char *) If not null. Specifies a description for the current action. Only the first `DITS_ACT_DESCR_LEN` (79) characters are used. Can be fetched using DitsGetActDescr(3) and is output by control messages such as `LISTACTALL` and `LISTACT`.

(!) status (StatusType *) Modified status.

Include files: DitsFix.h

External functions used:

strncpy	`CRTL`	Copy one string to another.

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine

See Also: The Dits Specification Document, DitsPutActions(3), DitsPutAction(3), DitsGetActDescr(3), DitsPutActDescr().

Support: Tony Farrell, `AAO`

C.132 DitsPutTransData — Associate an item with a transaction.

Function: Associate an item with a transaction.

Description: This call associates the specified data item with the specified transaction, which must have been returned by another dits call. The item can be retrived by DitsGetTransData().

Note, if the transaction was started in uface context, then the uface context code will be overriden by this call.

Note, this function is implemented as a macro.

Language: C

Call:
(Void) = DitsPutTransData(data,transid,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) data (void *) The data item to associated with the transaction

(>) transid (DitsTransIdType) The transaction as returned by another Dits call.

(!) status (StatusType *) Modified status.

Include files: DitsInteraction.h

External functions used: None

External values used: None

Prior requirements: None.

See Also: The Dits Specification Document, DitsGetTransData(3), DitsPutUserData(3), DitsPutPathData(3), DitsPutActData(3).

Support: Tony Farrell, `AAO`

C.133 DitsPutUserData — Put a user defined value in the task common block.

Function: Put a user defined value in the task common block.

Description: 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 varible to be saved and restored at each context switch. Although user written routines can use this library, its use makes user code VxWorks dependent and makes the task context switch time slower.

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 and for all the variables which would otherwise be static/global to be placed in the dymaically allocated area. This is the technique used by Dits to protect its common block

To avoid the user having to add another task variable, the DitsPutUserData 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 supply to DitsPutUserData the address of a dynamically allocated structure.

See the GitTpi routines for a way to access this item throughout packages.

Note, this function is implemented as a macro in C.

Language: C

Call:

(Void) = DitsPutUserData (data, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) data (void `*`) The data to store.

(`!`) status (StatusType `*`) Modified status.

Include files: DitsFix.h

External functions used: None

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine or a user interface response routine.

See Also: The Dits Specification Document, DitsGetUserData(3), DitsPutTransData(3), DitsPutPathData(3), DitsPutActData(3).

Support: Tony Farrell, `AAO`

C.134 DitsReleaseShared — Frees a shared memory section.

Function: Frees a shared memory section.

Description: Shared memory sections are used by `DITS` in the handling of bulk data transfers, generally initiated through a call to DitsInitMessBulk() or DitsTriggerBulk().

These shared memory sections are described in `DITS` using a structure called a 'shared memory information structure' of type DitsSharedMemInfoType. A program can map an existing shared memory section into its own address space, or can create a new one. The program that creates such a section should use DitsDefineShared() to set up the required shared memory information structure.

Once a program that has mapped a shared memory section no longer needs the section, it can release it, removing it from its memory space. To do that, it can call this routine.

Language: C.

Call:
DitsReleaseShared (SharedMemInfo, Delete, Status)

Parameters: ("`>`" input, "`<`" output, "`!`" modified)

(>) SharedMemInfo (DitsSharedMemInfoType *) The structure describing the shared memory section to be released.

(>) Delete (int) If passed as true (non-zero) the shared memory section will be flagged for deletion. When every task that has mapped such a shared memory section has released it, the section will be deleted. It is enough for one task to set this delete flag, and it is usually convenient if it is the task that created the section that flags it for deletion.

(!) Status (IMP_Status *) Inherited status value. If a non-zero value is passed, this task returns immediately. Otherwise, if an error is detected, an `IMP__` error code will be returned.

External variables used: None.

Include Files: DitsBulk.h

See Also: The DITS Specification Document, DitsDefineShared(3), DitsInitMessBulk(3), DitsTriggerBulk(3).

Prior requirements: None.

Support: Tony Farrell, AAO

Note: There may be a problem with setting the Delete flag for memory allocated using VxWorks, or other systems that do not check that all processes have released a memory block before freeing it. It may be that IMP itself needs to keep an access count for shared memory blocks.

C.135 DitsRequestNotify — Request a message when the buffers to a task empty.

Function: Request a message when the buffers to a task empty.

Description: This routine requests that we be notified when the buffers associated with a path to another task empty, enabling us to start sending messages again.

This routine can be called if DitsInitiateMessage/DitsObey etc. /ErsOut/MsgOut or DitsTrigger return a status of DITS__NOSPACE. In the later cases, you can use DitsGetParentPath to get the path.

It allows us to try to send the message again later.

Any more attempts to send messages on the path before the notify message is return will result in an error, with a status of DITS__NOTIFYWAIT

Language: C

Call:

(Void) = DitsRequestNotify (path, transid, status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) path (DitsPathType) Path to the task involved.

(<) transid (DitsTransIdType *) Transaction id for the transaction started.

(!) status (StatusType *) Modified status.

External functions used:

ImpRequestNotify	Imp	Request notification.
strlen	Crtl	Get the length of a string

External values used: DitsTask

Prior requirements: Can only be called from a Dits application routine or a user interface response routine

See Also: The Dits Specification Document, DitsPathGet(3), DitsAppInit(3), DitsGetPathSize(3), DitsGetMsgLength(3), Sds manual.

Support: Tony Farrell, AAO

C.136 DitsRestoreTask — Restore the interrupted Task Id

Function: Restore the interrupted Task Id

Description: Some Systems, such as VxWorks based systems, run all programs in a common address space. In such systems static and global variables can be seen by all tasks. To allow tasks to have private copies of static and global variables it is possible to have such variables saved and restored during task context switching. Dits uses this technique to store task specific information. In such systems it is sometimes necessary to call Dits routines outside the context of a task (say in an interrupt routine), during which the task specific information will be unavailable. This routine is used in conjunction with DitsGetTaskId and DitsEnableTask to make the task specific information available in such places.

This call is made in interrupt handlers. The argument should be the value returned by a previous call to DitsEnableTask in the interrupt handler. After this call is made, Dits routines can no longer be used.

Please see DitsEnableTask() for full details on using these routines.

Note, this function is implemented as a macro.

Language: C

Call:
(Void) = DitsRestoreTask (TaskId)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

(>) TaskId (DitsTaskIdType) A value returned by DitsEnableTask.

Include files: DitsSys.h

External functions used: None

External values used: DitsTask

Prior requirements: DitsAppInit(3) should have been called.

See Also: The Dits Specification Document, DitsEnableTask(3), DitsGetTaskId(3), DitsSignalByName(3), DitsSignalByIndex(3).

Support: Tony Farrell, AAO

C.137 DitsScanTasks — Scan the tasks known to DRAMA on current machine.

Function: Scan the tasks known to DRAMA on current machine.

Description: This routine scans the tasks known to DRAMA (IMP) on the current machine, returning details about each task. The user would normally call this routine multiple times until all tasks have been scanned or the required information obtained.

Language: C

Call:
(void) = DitsScanTasks (ScanInfo, TaskDetails, status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(!) ScanInfo (int *) The integer that this argument is the address of should be set to zero by the caller at the start of a scan through the tasks in the system. This routine will then set it to the index number of the oldest task on the system, which it supplies details for. Subsequent calls should pass this value as returned by the last call. Eventually, a call will return with this integer set back to zero, which indicates that all the tasks in the system have now been scanned.

(<) TaskDetails (DitsTaskDetailsType *) A structure that receives details of the task in question. See below for details.

(!) status (StatusType *) Modified status.

Include files: DitsInteraction.h

Fields set in TaskDetails by this call: TaskName(char[]), LocalTask(int), Translator(int), TaskType(int), TaskDescr(char[]), CurrentTask(int), Translated(int).

LocalTask is set true if the task is running on the local machine. Translator is set true if the task is a translator task (a specialised task used for communicating with non-IMP tasks). TaskType and TaskDescr are, respectively, an integer and character string optionally associated with the task through a call to DitsSetDetails(). If no such call has been made, TaskType is returned as zero and TaskDescr is set to a null string (one whose first character is a nul). CurrentTask is set true if the details obtained refer to the calling task itself. TaskName is the name under which the task registered. Translated is true if the task is not itself really part of the DRAMA system but is one that can be accessed through a translator task (normally, programs do not need to make any distinction between translated tasks and normal tasks - also note that this information is not available if the task in question is remote).

External functions used:

ImpGetDetails	Imp	Returns the details of a registered task

External values used: DitsTask - Details of the current task

Prior requirements: DitsAppInit must have been called.

See Also: The Dits Specification Document, DitsSetDetails(3), DitsGetTaskDescr(3), DitsFindTaskByType(3), DitsGetTaskType(3).

Support: Tony Farrell, AAO

C.138 DitsSdsList — Use MsgOut to output the contents of an Sds structure.

Function: Use MsgOut to output the contents of an Sds structure.

Description: Use MsgOut to output the contents of an Sds structure.

Language: C

Call:
(Void) = DitsSdsList(id, status)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(>) id (SdsIdType) The id of the SDS structure to output.

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

Support: Tony Farrell, AAO

C.139 DitsSetDebug — Enable/Disable Dits internal debugging.

Function: Enable/Disable Dits internal debugging.

Description: When enabled, Dits will output information at appropriate times to assist in debugging. See the flag argument for the logging modes available.

The argument is an integer which will be considered a mask of these flags.

You can also set debugging externally using the ditscmd -c option, with the name DEBUG or by setting the environment variable DITS_DEBUG to an appropriate value before starting the task.

Note that if a logging system is enabled (see DitsSetLogSys(3)) then logging is done by calling the logging system's logLogMsg routine with the level set to DITS_LOG_INTERNAL

Language: C

Call:
(Void) = DitsSetDebug (flag,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) flag (Int) Mask of values. To turn logging off, specify `DITS_LOG_OFF` - zero, only. Otherwise, specify a mask of the flags listed below

(`!`) status (StatusType `*`) Modified status.

Log flags:

`DITS_LOG_BASIC`	Basic logging of important events (Integer value 1)
`DITS_LOG_IMPSYS`	Dump details of `IMP` System message. (Integer value 2)
`DITS_LOG_IMPINIT`	Output details of calls to `IMP` (Integer value 4)
`DITS_LOG_MON`	Output monitor message details (Integer value 8)
`DITS_LOG_ACT`	Output action scheduling events. (Integer value 16)
`DITS_LOG_DETAILS`	Log details of such things as message sizes. (Integer value 32)
`DITS_LOG_BULK`	Log details of bulk data messages (Integer vaue 64)
`DITS_LOG_LIBS`	Usable by `DRAMA` libraries to log their own details, such as `DCPP` transaction details. (Integer value 128)
`DITS_LOG_IMPEVENTS`	Turn on `IMP` event logging (see `IMP` tasklog command for more info. When turned off or if this is on at task shutdown then the imp events log will be written to a file named after the task name and the file type .imp_log, otherwise you can use the `IMP` tasklog command to dump the log. (Integer value 256)
`DITS_LOG_RXLPEXIT`	Log exit from DitsMsgReceive, including time tags. (Integer value 4096)
`DITS_LOG_ALL`	Log everything available. (Integer value 65535)

Include files: DitsSys.h

External functions used: None

External values used: DitsTask

See Also: DitsLogMsg(3), DitsSetLogSys(3).

Prior requirements: DitsAppInit should have been called.

Support: Tony Farrell, AAO

C.140 DitsSetDetails — Set the details of this task.

Function: Set the details of this task.

Description: Once an task has called DitsAppInit, it can provide some additional information about itself that other tasks can access (through the enquiry routine DitsGetTaskType and DitsGetTaskDescr). This can make it easier for other tasks to decide if they want to try to communicate with this task. The IMP system allows a single integer type field and a character string to be associated with the current task using this routine. The meaning of these is entirely up to the task itself (or more usually, up the the designer of a set of cooperating tasks).

A task can also use DitsFindByType to find the name of a task of a given type.

Note, this function is implemented as a macro calling ImpSetDetails.

Language: C

Call:
(Void) = DitsSetDetails (TaskType,TaskDescr,status)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(>) TaskType (int) A user-defined integer value to be associated with the task.

(>) TaskDescr (const char *) A nul-terminated string to be associated with the task.

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used:

ImpSetDetails	Imp	Set task details.

External values used: DitsTask

Prior requirements: DitsAppInit(3) should have been called.

See Also: The Dits Specification Document, DitsScanTasks(3), DitsGetTaskDescr(3), DitsFindTaskByType(3), DitsGetTaskType(3).

Support: Tony Farrell, AAO

C.141 DitsSetFixFlags — Sets the value of the flags used to commuicate with the fixed part.

Function: Sets the value of the flags used to commuicate with the fixed part.

Description: This function allows some extra communication with the DRAMA fixed part above what is normally provided.

Language: C

Call:
(void) = DitsSetFixFlags(flags)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) flags (int - mask) A mask of flags. Possible values are

DITS_M_NO_SDS_CHECK	Don't do SDS leak check for this action, even if log flag is set - allows for actions which do allocate or release SDS ids on purpose and for lower level checking

Include files: DitsSys.h

Prior requirements: DitsAppInit() should have been invoked, must be in an action/uface context.

Support: Tony Farrell, AAO

C.142 DitsSetLogSys — Set the logging system to be used by DITS.

Function: Set the logging system to be used by DITS.

Description: DITS supports logging through an external log system. By default, no logger is provided. This routine can be used to provide a logging system or to inquire about the current enabled logging system, if any.

Language: C

Call:
(void) = DitsSetLogSys (newLogSys, oldLogSys, status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) newLogSys (DitsLogSysType *) The new logging system, see below.

(>) oldLogSys (DitsLogSysType *) The old logging system, see below.

(!) status (StatusType *) Modified status.

Include files: DitsUtil.h

Logging Systems: A logging system implements a number of routines which are invoked by `DRAMA` at various points to allow it to log operations. `DITS` does not provide such a logging system and it must be provided through this call (since efficency reaons and local requirements determine what logging should be done and how).

If the "newLogSys" argument is supplied, it is the address of a structure containing a client data item and a bunch of routine pointers. Particular ones of these routines are invoked at various points to log `DRAMA` operations. If any routine is null - the relevant items are not logged - so you can choose what you wish to log.

If the "oldLogSys" argument is supplied, it is the address of a structure into which the current log system details are placed.

Note that the logErs routine is passed down to Ers using ErsSetLogRoutine(). It is assumed that only `DRAMA` is changing the Ers log routine. Routines marked with (*) are not yet implemented.

It is suggested that you use memset() to zero the contents of this structure before setting the routines of interest. This will ensure upward compatbility of your code.

The are a large number of different routines since there are many points in `DRAMA` which you may want to log and they tend to pass different items which you may want to log.

Note that when these calls are invoked, then can use DitsGetContext() to get the current action context. If this is `NOT DITS_CTX_UFACE`, then they can use DitsGetName() to get the action name and DitsGetSeq() to get the action sequence counter value.

DitsLogSysType contains the following item.

clientData	A void * item which is passed to each log routine then invoked.
logActEnt	A routine invoked just before action entry and `UFACE` routine entry. Type DitsLogActEntRoutineType.
logActReturn	A routine invoked when an action/`UFACE` entry returns. Note that the request argument to this routine might be set to the value -1. This happens for a log made on entry to DitsActionWait(3)/DitsUfaceWait(3). Type DitsLogActReturnRoutineType.
logGetPath	A routine invoked when a GetPath operation is started (If the return status is bad then the status of this call is bad). Type DitsLogGetPathRoutineType.
logSignal	A routine invoked when a DitsSignal operation is performed. (If the return status is bad then the status of this call is bad). `NOTE`. Many `DRAMA` applications uses this from Interrupt/Unix Signal context. The logging routine must handle mode correctly. Type DitsLogGetPathRoutineType.
logMsgSend	A routine invoked when a message is sent to another task (If the return status is bad then the status of this call is bad). Type DitsLogSendRoutineType.
logLoad	A routine invoked when a load operation is initiated (If the return status is bad then the status of this call is bad). Type DitsLogLoadRoutineType.
logNotify	A routine invoked when a GetNotify operation is initiated. Type DitsLogNotifyReqRoutineType.
logTrigger	A routine invoked after sending a trigger message. Type DitsLogTriggerRoutineType.
logTrigBulk	A routine invoked after sending a bulk data trigger message. Type DitsLogTrigerBulkRoutineType.
logBulkMsg	A routine invoked after sending a bulk data message. type DitsLogBulkSendRoutineType.
logMsgOut	A routine invoked to log MsgOut messages. Type DitsLogMsgOutRoutineType.
logErs	A routine invoked to log Ers reports. Type ErsLogRoutineType (defined by Ers.h).
logLogMsg	A routine used to implement DitsLogMsg(). Will also be invoked when `DRAMA`'s internal logging is enabled (see DitsSetDebug(3)) with a level of `DITS_LOG_INTERNAL`. Type DitsLogLogMsgRoutineType.
logShutdown	A routine invoked when the task is shutting down. In addition to giving the ability to log task shutdown, allows the logging system to be tidied up. It will never be invoked again by `DRAMA` after this call. Type DitsLogShutdownRoutineType.
logFlush	Invoked

Function Prototypes Used: typedef DVOID (*DitsLogActEntRoutineType)(DVOIDP client_data, StatusType *status);

typedef DVOID (*DitsLogActReturnRoutineType)(DVOIDP client_data, int argDelete, SdsIdType argOut, DitsReqType request, int delaySet, DCONSTV DitsDeltaTimeType * delay, StatusType exitStatus, StatusType *status);

typedef DVOID (*DitsLogGetPathRoutineType)(DVOIDP client_data, DCONSTV char * TaskName, DCONSTV char * node, int pathFlags, DCONSTV DVOIDP pathInfo, DitsPathType path, DitsTransIdType transid, StatusType * status);

typedef DVOID (*DitsLogSignalRoutineType)(DVOIDP client_data, long int actptr, DCONSTV DVOIDP ptr, SdsIdType argSignal, StatusType *status);

typedef DVOID (*DitsLogMsgSendRoutineType)(DVOIDP client_data, long int sendflags, DitsPathType path, DitsTransIdType transid, DCONSTV DitsGsokMessageType * message, StatusType *status);

typedef DVOID (*DitsLogLoadRoutineType)(DVOIDP client_data, DCONSTV char * node, DCONSTV char * TaskName, DCONSTV char * loadArgString, long int loadFlags, DCONSTV DitsTaskParamType * TaskParams, DitsTransIdType transid, StatusType *status);

typedef DVOID (*DitsLogNotifyReqRoutineType)(DVOIDP client_data, int notifyRequested, DitsPathType path, DitsTransIdType transid, StatusType *status);

typedef DVOID (*DitsLogTriggerRoutineType)(DVOIDP client_data, SdsIdType triggerArg, StatusType *status);

typedef DVOID (*DitsLogTrigBulkRoutineType)(DVOIDP client_data, DCONSTV DVOIDP SharedMemInfo, int sds, int NotifyBytes, DitsTransIdType transid, StatusType *status);

typedef DVOID (*DitsLogBulkSendRoutineType)(DVOIDP client_data, DCONSTV long int BulkFlags, DCONSTV DitsPathType path, DCONSTV DVOIDP SharedMemInfo, int NotifyBytes, DitsTransIdType transid, DCONSTV DitsGsokMessageType * message, StatusType * status);

typedef DVOID (*DitsLogMsgOutRoutineType)(DVOIDP client_data, DCONSTV char * messageText, StatusType *status);

typedef DVOID (*DitsLogLogMsgRoutineType)(DVOIDP client_data, unsigned level, DCONSTV char * prefix, DCONSTV char * fmt, va_list ap, StatusType *status);

typedef DVOID (*DitsLogShutdownRoutineType)(DVOIDP client_data, int taskShutdown, StatusType * status);

typedef DVOID (*ErsLogRoutineType)(DVOIDP client_data, DCONSTV ErsMessageType * errorReport, StatusType * status);

typedef DVOID (*DitsLogFlushRoutineType)(DVOIDP client_data, StatusType * status);

typedef DVOID (*DitsLogInfoRoutineType)(DVOIDP client_data, StatusType * status);

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) client_data (void *) The clientData item found in the newLogSys structure.

(>) argDelete (int) If argOut is valid (non-zero), then this flag is true if DITS will be deleting the argument when processing of this message completes.

(>) **argOut (SdsIdType)** The `SDS ID` of any action/message output argument (which is to be returned to the initiator of this message). Zero if there is not any.

(>) **request (DitsReqType)** The action request - see DitsPutRequest.

(>) **delaySet (int)** Has the action set a delay with DitPutDelay.

(>) **delay (const DitsDeltaTimeType)** The delay, if set. Otherwise may be a null pointer.

(>) **exitStatus (StatusType)** The status the action exited with.

(>) **TaskName (const char *)** The name of the task. For loading, the name of the program to be loadded.

(>) **node (const char *)** The name of the node. Note, may be a null pointer or zero length string if no node name is specified.

(>) **pathFlags (int)** The flags to a PathGet operations. See DitsPathGet().

(>) **pathInfo (void *)** Details passed ot DitsPathGet in its info structure. Only valid if `NOT` an inquiry, (transid `!=` 0) in which case it can be cast to a (DitsPathInfoType *).

(>) **transid (int)** The transaction id.

(>) **actptr (int)** The index of the action being signaled.

(>) **ptr (const void *)** A poonter value passed to a signalling routine.

(>) **argSignal (SdsIdType)** The Argument to a signalling routine.

(>) **sendFlags (int)** The send flags.

(>) **message (const DitsGsokMessageType *)** The message sent.

(>) **loadArgString (const char *)** The load argument string.

(>) **loadFlags (long)** Load flags.

(>) **TaskParams (const DitsTaskParamType *)** The task load parameters.

(>) **triggerArg (SdsIdType)** The agument to the trigger message.

(>) **SharedMemInfo (void *)** Details of shared memory.

(>) **sds (int)** Does thee shared memory contain `SDS`.

(>) **NotifyBytes (int)** Bulk data notification interval.

(>) **BulkFlags (long)** Flags to a bulk data send

(>) **messageText (cons char *)** The message text to be output.

(>) **level (unsigned)** Logging level - see DitsLogMsg() flag with the possible addition of `DITS_LOG_INTERNAL`.

(>) **prefix (const char *)** Prefix string to DitsLogMsg().

(>) **fmt (const char *)** A C printf style formating string.

(>) **ap (va_list)** Argument list for a C printf style formating string.

(>) **errorReport (const ErsMessageType *)** Details of an error report.

(>) **status (StatusType *)** Modified status. See the DitsLogSysType notes above for details on status usage for each function.

External functions used:

External values used: DitsTask

Prior requirements: Can only be called from a Dits Application routine

See Also: The Dits Specification Document. DitsLogMsg(3), ErsSetLogRoutine(3), ErsStart(3), DitsSetDebug(3).

Support: Tony Farrell, `AAO`

C.143 DitsSetParam — Send a 'Set parameter' message to a task

Function: Send a 'Set parameter' message to a task

Description: A `SET` message with the specified parameter name and argument is sent to the task on the given path.

If the message fails, a "transaction failure" message will be received by the task with the transaction id returned by this routine, otherwise a completion message will be received.

Language: C

Call:

(Void) = DitsSetParam(path,param,argument,transid, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) path (DitsPathType) The path to the task to which the message is directed. See DitsGetPath for details on how to get a path to another task.

(>) param (char *) The null terminated name of the parameter who's value is to be set.

(>) argument (SdsIdType) The value to set the parameter to. This is an Sds Id describing the value to which the parameter will be set. For details of its format, see the details of the parameter system routines being used by the target task.

(<) transid (DitsTransId *) Transaction id for the transaction started. If an address of zero is supplied, then don't create a transaction id. In this case we will never be notified of transaction errors

(!) status (StatusType *) Modified status.

Include files: DitsParam.h

Prior requirements: Can only be called from a Dits application routine or when the context is `DITS_CTX_UFACE`.

See Also: The Dits Specification Document, DitsInitateMessage(3), DitsGetParam(3), DitsAppParamSys(3), DitsSetParamSetup(3), DitsSetParamTidy(3), DuiExecuteCmd(3), DulMessageW(3).

Support: Tony Farrell, `AAO`

C.144 DitsSetParamSetup — Set up a DitsGsokMessageType variable for a set message.

Function: Set up a DitsGsokMessageType variable for a set message.

Description: Given a parameter name and value, this routine set's up a DitsGsokMessageType variable for `SET` parameter message, transparently handling long parameter name.

You should invoke DitsSetParamTidy for the same DitsGsokMessageType variable after sending this message.

Language: C

Call:
(Void) = DitsSetParamSetup(param,argument,msg, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) param (char *) The null terminated name of the parameter who's value is to be set.

(>) argument (SdsIdType) The value to set the parameter to. This is an Sds Id describing the value to which the parameter will be set. For details of its format, see the details of the parameter system routines being used by the target task.

(<) msg (DitsGsokMessageType *) The message structure which will be set up.

(!) status (StatusType *) Modified status.

Include files: DitsInteract.h

Prior requirements: None

See Also: The Dits Specification Document, DitsInitateMessage(3), DitsSetParam(3), DitsSetParamTidy(3), DitsAppParamSys(3).

Support: Tony Farrell, `AAO`

C.145 DitsSetParamTidy — Tidy up after a call to DitsSetParamSetup.

Function: Tidy up after a call to DitsSetParamSetup.

Description: This routine should be called after the message set up with DitsSetParamTidy has been sent to tidy up after that call.

Language: C

Call:
(Void) = DitsSetParamTidy(msg, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) msg (DitsGsokMessageType *) The message structure.

(!) status (StatusType *) Modified status.

Include files: DitsInteract.h

Prior requirements: None

See Also: The Dits Specification Document, DitsInitateMessage(3), DitsSetParam(3), DitsSetParamSetup(3), DitsAppParamSys(3).

Support: Tony Farrell, AAO

C.146 DitsSignalByIndex — Trigger the rescheduling of an action.

Function: Trigger the rescheduling of an action.

Description: The routine allows an action to trigger the rescheduling of another action in the current task.

Although this could in theory be done by DitsKick, by using DitsSignal series you avoid having to have a path to this task. In addition, this routine can be used when there is no action context (such as in an interrupt serivce routine).

The target action may be awaiting rescheduling for any reason (having put a request of DITS_REQ_WAIT, DITS_REQ_SLEEP, DITS_REQ_MESSAGE or DITS_REQ_STAGE).

Since this routine is often used for communication between interrupt routines and main line code, the message is sent as a high priority message and will thus be put a the begining of the incomming message buffer.

In the case of interrupt service routines, you probably need to first enable the appropriate Dits task. See DitsGetTaskId(), DitsEnableTask() and DitsRestoreTask().

Note, as this routine is often used to allow non-DRAMA parts of a program to communicate with a DRAMA program, applications often get into trouble if DitsSignalByIndex/DitsSignalByName fail. I recommend you always check status after such a call and report errors using an appropriate system error routine. For VxWorks interrupt handlers, this is the logMsg() call. For WIN32, this is probably the MessageBox() call. For most others it is fprintf() to stderr. The most common error is to fail to set status to STATUS__OK before calling these routines.

WARNING 1: If your are sending signals from an ISR routine and sending messages to your own task from main line code, you should specify the flag DITS_M_SEP_SIG_BUF when you call DitsAppInit(). A failure to do this could result in buffer corruption.

DO NOT (in the current version) send messages using DitsSignal() from both an ISR and main line code if they could occur at the same time. (A later version may fix this if thereis demand).

WARNING 2: Under VxWorks, do not call DitsSignal() from and ISR or another task if your task has specified the DITS_M_X_COMPATIBLE. On VxWorks, there is a rare case using this

notification method which may cause blocking. A warning will be output to the VxWorks console if you are doing this.

Note, if you want to attach a non-SDS item to your message, use DitsSignalByNamePtr(3) or DitsSignalByIndexPtr(3)

Language: C

Call:
(Void) = DitsSignalByIndex (index, argument, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) index (long int) The index to the action to kick. This must be a value returned by DitsGetActIndex().

(>) argument (SdsIdType) An argument to the action. This should be an Sds id. See DitsGetArgument for more details on action arguments. Note that for this routine, no copy is made of this argument structure - this argument structure will be deleted after the action is invoked. Note that in previous versions or `DRAMA`, you could use this item to pass simple integer items, assuming the invoking action used DitsArgNoDel(3). This behaviour is no longer supported and DitsArgNoDel(3) has been removed. SEe DitsSignalByIndexPtr(3) for an alternative.

(!) status (StatusType *) Modified status.

Include files: DitsSignal.h

External functions used:

Dits___SendTap	Dits internal	Send a tap message

External values used: DitsTask

Prior requirements: Can only be called from a Dits Application routine

See Also: The Dits Specification Document, DitsKillByIndex(3), DitsSignalByName(3), DitsGetActIndex(3), DulIntInit(3), DulIntSignal(3), DitsSignalByIndexPtr(3), DitsSignalByNamePtr(3).

Support: Tony Farrell, `AAO`

C.147 DitsSignalByIndexPtr — Trigger the rescheduling of an action.

Function: Trigger the rescheduling of an action.

Description: The routine allows an action to trigger the rescheduling of another action in the current task.

Although this could in theory be done by DitsKick, by using DitsSignal series you avoid having to have a path to this task. In addition, this routine can be used when there is no action context (such as in an interrupt serivce routine).

The target action may be awaiting rescheduling for any reason (having put a request of `DITS_REQ_WAIT`, `DITS_REQ_SLEEP`, `DITS_REQ_MESSAGE` or `DITS_REQ_STAGE`).

Since this routine is often used for communication between interrupt routines and main line code, the message is sent as a high priority message and will thus be put a the begining of the incomming message buffer.

In the case of interrupt service routines, you probably need to first enable the appropriate Dits task. See DitsGetTaskId(), DitsEnableTask() and DitsRestoreTask().

Note, as this routine is often used to allow non-DRAMA parts of a program to communicate with a `DRAMA` program, applications often get into trouble if DitsSignal... fail. I recommend you always check status after such a call and report errors using an appropriate system error routine. For VxWorks interrupt handlers, this is the logMsg() call. For `WIN32`, this is probably the MessageBox() call. For most others it is fprintf() to stderr. The most common error is to fail to set status to `STATUS__OK` before calling these routines.

`WARNING` 1: If your are sending signals from an `ISR` routine and sending messages to your own task from main line code, you should specify the flag `DITS_M_SEP_SIG_BUF` when you call DitsAppInit(). A failure to do this could result in buffer corruption.

`DO NOT` (in the current version) send messages using DitsSignal() from both an `ISR` and main line code if they could occur at the same time. (A later version may fix this if there is demand).

`WARNING` 2: Under VxWorks, do not call DitsSignal() from and `ISR` or another task if your task has specified the `DITS_M_X_COMPATIBLE`. On VxWorks, there is a rare case using this notification method which may cause blocking. A warning will be output to the VxWorks console if you are doing this.

Also see DitsSignalByName().

Language: C

Call:

(Void) = DitsSignalByIndexPtr (index, argument, sigArg, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) index (long int) The index to the action to kick. This must be a value returned by DitsGetActIndex().

(>) argument (SdsIdType) An argument to the action. This should be an Sds id. See DitsGetArgument for more details on action arguments. Note that for this routine, no copy is made of this argument structure - this argument structure will be deleted after the action is invoked.

(>) **sigArg (void *)** A pointer which will be assoicated with the signal. The action can fetch this using DitsGetSigArg(3).

(!) **status (StatusType *)** Modified status.

Include files: DitsSignal.h

External functions used:

Dits___SendTap	Dits internal	Send a tap message

External values used: DitsTask

Prior requirements: Can only be called from a Dits Application routine

See Also: The Dits Specification Document, DitsKillByIndex(3), DitsSignalByNamePtr(3), DitsGetActIndex(3), DulIntInit(3), DulIntSignalByIndex(3), DitsGetSigArg(3).

Support: Tony Farrell, AAO

C.148 DitsSignalByName — Trigger the rescheduling of an action.

Function: Trigger the rescheduling of an action.

Description: The routine allows an action to trigger the rescheduling of another action in the current task.

Although this could in theory be done by DitsKick, by using DitsSignal series you avoid having to have a path to this task. In addition, this routine can be used when there is no action context (such as in an interrupt serivce routine).

The target action may be awaiting rescheduling for any reason (having put a request of DITS_REQ_WAIT, DITS_REQ_SLEEP, DITS_REQ_MESSAGE or DITS_REQ_STAGE).

Since this routine is often used for communication between interrupt routines and main line code, the message is sent as a high priority message and will thus be put a the begining of the incomming message buffer.

In the case of interrupt service routines, you probably need to first enable the appropriate Dits task. See DitsGetTaskId(), DitsEnableTask() and DitsRestoreTask().

Note, as this routine is often used to allow non-DRAMA parts of a program to communicate with a DRAMA program, applications often get into trouble if DitsSignalByIndex/DitsSignalByName fail. I recommend you always check status after such a call and report errors using an appropriate system error routine. For VxWorks interrupt handlers, this is the logMsg() call. For WIN32, this is probably the MessageBox() call. For most others it is fprintf() to stderr. The most common error is to fail to set status to STATUS__OK before calling these routines.

`WARNING` 1: If your are sending signals from an `ISR` routine and sending messages to your own task from main line code, you should specify the flag `DITS_M_SEP_SIG_BUF` when you call DitsAppInit(). A failure to do this could result in buffer corruption.

`DO NOT` (in the current version) send messages using DitsSignal() from both an `ISR` and main line code if they could occur at the same time. (A later version may fix this if there is demand).

`WARNING` 2: Under VxWorks, do not call DitsSignal() from and `ISR` or another task if your task has specified the `DITS_M_X_COMPATIBLE`. On VxWorks, there is a rare case using this notification method which may cause blocking. A warning will be output to the VxWorks console if you are doing this.

Note, if you want ot attach a non-SDS item to your message, use DitsSignalByNamePtr(3) or DitsSignalByIndexPtr(3)

Language: C

Call:

(Void) = DitsSignalByName (name, argument, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) name (Char *) The null terminated name of the action to signal

(>) argument (SdsIdType) An argument to the action. This should be an Sds id. See DitsGetArgument for more details on action arguments. Note that for this routine, no copy is made of this argument structure - this argument structure will be deleted after the action is invoked. Note that in previous versions or `DRAMA`, you could use this item to pass simple integer items, assuming the invoking action used DitsArgNoDel(3). This behaviour is no longer supported and DitsArgNoDel(3) has been removed. See DitsSignalByNamePtr(3) for an alternative.

(!) status (StatusType *) Modified status.

Include files: DitsSignal.h

External functions used:

Dits___ActptrByName	Dits internal	Return the index to an action
DitsSignalByIndex	Dits	Signal by index.

External values used: DitsTask

Prior requirements: Can only be called from a Dits Application routine

See Also: The Dits Specification Document, DitsKillByName(3), DitsSignalByIndex(3), DulIntInit(3), DulIntSignal(3), DitsSignalByNamePtr(3), DitsSignalByIndex(3).

Support: Tony Farrell, `AAO`

C.149 DitsSignalByNamePtr — Trigger the rescheduling of an action.

Function: Trigger the rescheduling of an action.

Description: The routine allows an action to trigger the rescheduling of another action in the current task.

Although this could in theory be done by DitsKick, by using DitsSignal Series you avoid having to have a path to this task. In addition, this routine can be used when there is no action context (such as in an interrupt serivce routine).

The target action may be awaiting rescheduling for any reason (having put a request of `DITS_REQ_WAIT`, `DITS_REQ_SLEEP`, `DITS_REQ_MESSAGE` or `DITS_REQ_STAGE`).

Since this routine is often used for communication between interrupt routines and main line code, the message is sent as a high priority message and will thus be put a the begining of the incomming message buffer.

In the case of interrupt service routines, you probably need to first enable the appropriate Dits task. See DitsGetTaskId(), DitsEnableTask() and DitsRestoreTask().

Note, as this routine is often used to allow non-DRAMA parts of a program to communicate with a `DRAMA` program, applications often get into trouble if DitsSignalByIndex/DitsSignalByName fail. I recommend you always check status after such a call and report errors using an appropriate system error routine. For VxWorks interrupt handlers, this is the logMsg() call. For `WIN32`, this is probably the MessageBox() call. For most others it is fprintf() to stderr. The most common error is to fail to set status to `STATUS__OK` before calling these routines.

`WARNING` 1: If your are sending signals from an `ISR` routine and sending messages to your own task from main line code, you should specify the flag `DITS_M_SEP_SIG_BUF` when you call DitsAppInit(). A failure to do this could result in buffer corruption.

`DO NOT` (in the current version) send messages using DitsSignal() from both an `ISR` and main line code if they could occur at the same time. (A later version may fix this if there is demand).

`WARNING` 2: Under VxWorks, do not call DitsSignal() from and `ISR` or another task if your task has specified the `DITS_M_X_COMPATIBLE`. On VxWorks, there is a rare case using this notification method which may cause blocking. A warning will be output to the VxWorks console if you are doing this.

Language: C

Call:

(Void) = DitsSignalByNamePtr (name, argument, sigArg, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) name (Char *) The null terminated name of the action to signal

(>) argument (SdsIdType) An argument to the action. This should be an Sds id. See DitsGetArgument for more details on action arguments. Note that for this routine, no copy is made of this argument structure - this argument structure will be deleted after the action is invoked.

(>) sigArg (void *) A pointer which will be assoicated with the signal. The action can fetch this using DitsGetSigArg(3).

(!) status (StatusType *) Modified status.

Include files: DitsSignal.h

External functions used:

Dits___ActptrByName	Dits internal	Return the index to an action
DitsSignalByIndex	Dits	Signal by index.

External values used: DitsTask

Prior requirements: Can only be called from a Dits Application routine

See Also: The Dits Specification Document, DitsKillByName(3), DitsSignalByIndexPtr(3), DulIntInit(3), DulIntSignalByName(3).

Support: Tony Farrell, AAO

C.150 DitsSignalDrama2 — Trigger the rescheduling of an action from DRAMA 2.

Function: Trigger the rescheduling of an action from DRAMA 2.

Description: The routine allows an action to trigger the rescheduling of another action in the current task.

This signal is intended only for use by the DRAMA 2 API. It triggers a special action reschedule event. (Reschedule Message type of DITS_MSG___SIGNAL_DRAMA2.

Language: C

Call:
(Void) = DitsSignalDrama2 (index, status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) index (long int) The index to the action to kick. This must be a value returned by DitsGetActIndex().

(!) status (StatusType *) Modified status.

Include files: DitsSignal.h

External functions used:

Dits___SendTap	Dits internal	Send a tap message

External values used: DitsTask

Support: Tony Farrell, AAO

C.151 DitsSignalExit — Trigger the DRAMA Main loop to exit.

Function: Trigger the DRAMA Main loop to exit.

Description: The routine allows trigger the DRAMA Main-loop to exit. This routine might be invoked from an interrupt handler or thread, any separate thread of control from the DRAMA message processing loop. It sends an internal DRAMA message which is used to cause the main loop to exit.

Since this routine is often used for communication between interrupt routines and main line code, the message is sent as a high priority message and will thus be put a the begining of the incomming message buffer.

In the case of interrupt service routines, you probably need to first enable the appropriate Dits task. See DitsGetTaskId(), DitsEnableTask() and DitsRestoreTask().

WARNING 1: If your are sending signals from an ISR routine and sending messages to your own task from main line code, you should specify the flag DITS_M_SEP_SIG_BUF when you call DitsAppInit(). A failure to do this could result in buffer corruption.

DO NOT (in the current version) send messages using DitsSignal() from both an ISR and main line code if they could occur at the same time. (A later version may fix this if there is demand).

WARNING 2: Under VxWorks, do not call DitsSignal() from and ISR or another task if your task has specified the DITS_M_X_COMPATIBLE. On VxWorks, there is a rare case using this notification method which may cause blocking. A warning will be output to the VxWorks console if you are doing this.

Language: C

Call:
(Void) = DitsSignalExit (name, argument, status)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(>) exitStatus (StatusType) The main loop should exit with this status.

(!) status (StatusType *) Modified status.

Include files: DitsSignal.h

See Also: The Dits Specification Document. DitsSignalByIndex(3).

Support: Tony Farrell, AAO

C.152 DitsSpawnKickArg — Create an argument structure used when kick actions which spawn.

Function: Create an argument structure used when kick actions which spawn.

Description: Actions which spawn (allowing multiple actions of the same name) must be kicked by specifing an argument structure which allows the target task to determine which invocation of the action should be kicked.

This can be done by either specifing the action index (which the subsidary task can get using DitsGetActIndex()) as an argument named "KickByIndex" or anther task using the transaction id (as known by the parent action of this action), as wraped up in an argument by this call.

Note, arguments to the kick itself can be added to the argument created here, using standard ArgPut functions. Also, it is possible to change the transaction id in this structure using DitsSpawnKickArgUpdate().

Language: C

Call:
(void) = DitsSpawnKickArg (transid, arg, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) transid (DitsTransIdType) The transaction id returned when the obey was started.

(<) arg (SdsIdType *) The Sds id of a newly created sds item is returned here. Its first item consists of details of the transaction id acceptable for kicking the obey. You should delete the sds item when finished.

(!) status (StatusType *) Modified status.

Include files: DitsInteraction.h

External functions used:

ArgNew	Arg	Create a new argument structure.
SdsNew	Sds	Create a new Sds item
SdsPut	Sds	Put sds data values.
memset	Crtl	Set byte values.

External values used: none

Prior requirements: none

See Also: The Dits Specification Document, DitsGetActIndex(3), DitsSpawnKickArgUpdat3(3), DitsPutActions(3).

Support: Tony Farrell, `AAO`

C.153 DitsSpawnKickArgUpdate — Update an argument structure used when kick actions which spawn.

Function: Update an argument structure used when kick actions which spawn.

Description: Actions which spawn (allowing multiple actions of the same name) must be kicked by specifing an argument structure which allows the target task to determine which invocation of the action should be kicked.

This can be done by either specifing the action index (which the subsidary task can get using DitsGetActIndex()) as an argument named "KickByIndex" or anther task using the transaction id (as known by the parent action of this action), as wraped up in an argument by this call. This routine can be used to change the transaction id in such argument structure.

It can also be used to add such and item to an existing structure, which it does if it does not already exist within the structure. Note that in this case, if an item named "KickByIndex" is found, it is deleted. The add if is does not exist feature only works for internal `SDS` items, whilst the update only feature will also work for external `SDS` items.

Language: C

Call:

(void) = DitsSpawnKickArgUpdate (transid, arg, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) transid (DitsTransIdType) The transaction id returned when the obey was started.

(>) arg (SdsIdType) The Sds id of a structure which has been created by DitsSpawnKickArg.

(!) status (StatusType *) Modified status.

Include files: DitsInteraction.h

External functions used:

SdsFind	Sds	Find an Sds item.
SdsPut	Sds	Put sds data values.
memset	Crtl	Set byte values.

External values used: none

Prior requirements: none

See Also: The Dits Specification Document, DitsGetActIndex(3), DitsPutActions(3), DitsSpawnKickArg(3).

Support: Tony Farrell, `AAO`

C.154 DitsStop — Shutdown Dits.

Function: Shutdown Dits.

Description: Shutdown the messages system and deallocate memory used by the task common block.

This function is acutally implemented as a macro calling Dits___Stop.

Language: C

Call:
(int) = DitsStop (name,status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) name (char *) A name to use in error messages. This is normally the name supplied to DitsAppInit as the taskname and is required as this routine may be called after DitsAppInit has failed and the taskname supplied to it has not been recorded.

(>) status (StatusType *) Modified status. If status is Not STATUS__OK and the operating system is not VMS, then the text associated with the code is output. Under VMS, we rely on VMS outputting the code when the Main function returns or exit() is called with the status of this call.

Function Value: An appropriate status value for passing to the operating system using the function exit() or as the return value from the main function.

Include files: DitsSys.h

External functions used:

Dits___ExHandCancel	Dits internal	Cancel exit handlers.
Dits___TransIdDelete	Dits internal	Delete a transaction id.
Dits___PathDelete	Dits internal	Delete paths.
ImpDetach	IMP	Detach from message system.
SdsDelete	Sds	Delete an Sds item.
SdsFreeId	Sds	Free an Sds id.
free	CRTL	Free Allocated memory
taskVarDelete	VxWorks	When running under VxWorks only, delete a variable from the task's switch block.

External values used: DitsTask - Details of the current task

Prior requirements: DitsAppInit(3) should have been called

See Also: The Dits Specification Document, DitsAppInit(3).

Support: Tony Farrell, AAO

C.155 DitsTakeOrphans — Take over orphans.

Function: Take over orphans.

Description: The current action will either take over all outstanding orphan transactions or future orphan transactions or both.

An action which takes over future orphan transactions can stop doing so only by either exiting or by another action taking them over.

Once an action takes over an orphan transaction, that action will be invoked for messages related to that transaction and it appears as if the transaction belongs to the task which took it over.

Language: C

Call:
(void) = DitsTakOrphans (mode,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) mode (DitsOrphanModeTyp) One of

`DITS_TAKE_CURRENT`	Take ownership of current orphan transactions.
`DITS_TAKE_FUTURE`	Take ownership of future orphan transactions.
`DITS_TAKE_BOTH`	Take ownership of both current and future orphan transactions.

(!) status (StatusType *) Modified status.

Include files: DitsOrphan.h

External functions used:

Dits___TransIdRemove	Dits internal	Remove a transaction from a list

External values used: DitsTask

Prior requirements: DitsAppInit must have been called

See Also: The Dits Specification Document, DitsCheckTransactions(3), DitsIsOrphan(3), DitsPutOrphanHandler(3), DitsForget(3).

Support: Tony Farrell, `AAO`

C.156 DitsTaskFromPath — Return the name of a task given a path to it.

Function: Return the name of a task given a path to it.

Description: Given the path to a task, return the name of that task.

Note, this function is implemented as a macro.

Language: C

Call:
(Void) = DitsTaskFromPath(path,namelen,name,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) path (DitsPathType) The path

(>) namelen (Int) The size of name in bytes, should be at least `DITS_C_NAMELEN` bytes (20 bytes).

(>) name (char *) The name of the task is copied here. It will be null terminated, assuming the name can fit in.

(!) status (StatusType *) Modified status. Possible failure codes are

Include files: DitsInteraction.h Additional interface under C++ (const char *) = DitsTaskFromPath(path)

External functions used: None

External values used: None

Prior requirements: Can only be called from a Dits application routine

See Also: The Dits Specification Document, DitsPathGet(3), DitsAppInit(3).

Support: Tony Farrell, `AAO`

C.157 DitsTaskIsLocal — Indicates if a task on a given path is local

Function: Indicates if a task on a given path is local

Description: This function returns true if the task the path to which is specified is local. This value of this function is unspecified if the path specified is invalid.

Note, this function is implemented as a macro.

Language: C

Call:
(int) = DitsPathIsLocal(path)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

(>) path (DitsPathType) The path to check.

Function Value: True or false

Include files: DitsInteraction.h

External functions used: None

External values used: None

Prior requirements: None

See Also: The Dits Specification Document, DitsPathGet(3).

Support: Tony Farrell, AAO

C.158 DitsTrigger — Trigger the parent action

Function: Trigger the parent action

Description: This routine triggers the parent action of the current action to enter assuming it has an interest in trigger messages (see DitsInterested and DitsNotInterested).

The parent action is that action (probably in another task) which sent the OBEY message which initiated this action.

Note, this function is implemented using a macro to invoke Dits___SendTap.

Language: C

Call:
(Void) = DitsTrigger(argument,status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

(>) argument (SdsIdType) An argument to pass to the parent. This should be an Sds id. See DitsGetArgument and DitsPutArgument for more details on action arguments.

(!) status (StatusType *) Modified status.

Include files: DitsInteraction.h

External functions used:

Dits___SendTap	Dits internal	Send a message to the originator of the current action.

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits application routine

See Also: The Dits Specification Document, DitsInitiateMessage(3), DitsGetEntInfo(3).

Support: Tony Farrell, `AAO`

C.159 DitsTriggerBulk — Trigger parent action, sending a bulk data argument.

Function: Trigger parent action, sending a bulk data argument.

Description: This routine triggers the parent action of the current action to enter assuming it has an interest in trigger messages (see DitsInterested(3) and DitsNotInterested(3)).

The parent action is that action (probably in another task) which sent the `OBEY` message which initiated this action.

In this version of DitsTrigger(3), you can attach an amount of bulk data defined by an `DITS` shared memory segment. The bulk data is passed as efficiently as possible to the target task.

If the bulk data contains an exported `SDS` item, then the target task can receive the bulk data without doing any more work. Alternatively, and required if `SDS` is not involved, the target task can use various `DITS` inquiry functions to handle it in a specially.

Sending of bulk data involves a transaction and this routine returns a transaction id. The sending action may receive additional reschedule events with an entry reason of `DITS_REA_BULK_TRANSFERRED` indicating the progress of the transfer. It will also receive an entry with a reason of `DITS_REA_BULK_DONE` when the target task releases the shared memory. You can then call DitsGetEntBulkInfo() to retrieve details of the transfer.

The reception of `DITS_REA_BULK_TRANSFERRED` messages are dependent on the behavour of the target task. If it is a local task, then these mesages are recevid each time it calls DitsBulkArgReport(). If the target task is remote, these messages are sent by the local `IMP` transmitter task at a rate determined by it. If NotifyBytes is non-zero it indicates a hint to the target target about the rate at which these nofications are requested.

You should not delete the shared memory yourself until get the first message using one of these codes. When you get `DITS_REA_BULK_DONE` you are free to reuse all the shared memory as the target task has finished accessing the shared memory. You get reuse parts of the shared memory earily by making use of information retreived by DitsGetEntBulkInfo() to determine what parts of the bulk data have been used.

If the target task is local, then `DITS_REA_BULK_DONE` is received when the target task action entry returns or if the task invoked DitsBulkArgInfo(3), when it has called DitsBulkArgRelease(3).

If the target task is remote, then `DITS_REA_BULK_DONE` is recevied when the local transmitter task has finished forwarding the data to the remote machine.

As a result, in the remote task case, you are not notified when the target task has finished accessing the data, just when it is safe for you to re-use the local shared memory.

Language: C

Call:
(void) = DitsTriggerBulk(SharedMemInfo,sds,NotifyBytes,transid,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) SharedMemInfo (DitsSharedMemInfoType *) A structure describing the shared memory section containing the bulk data to be sent. This is set up with the routine DitsDefineShared(3) You should not delete this shared memory until you receive a response to this transaction with code `DITS_REA_BULK_TRANSFERRED` or `DITS_REA_BULK_DONE`.

(>) sds (int) Set this true (non-zero) if the shared memory item contains an exported `SDS` item. If true, the target task can access the item as an argument using `SDS` in the normal way (see DitsGetArgument(3). If true, this flag causes the target task to pass the shared memory address to the routine SdsAccess(3). If false, the shared memory contains data in an application private format and `DRAMA` makes no attempt to interpret it.

(>) NotifyBytes (int) If non-zero, it indicates your action should be notified (using `DITS_REA_BULK_TRANSFERRED` entries) every time the specified number of bytes are transferred. Note the this effect is somewhat dependent on the target task behaviour and you may only receive the final `DITS_REA_BULK_DONE` message.

(<) transid (DitsTransIdType *) Transaction id for the transaction started.

(!) status (StatusType *) Modified status.

Include files: DitsBulk.h

See Also: DitsTrigger(3), DitsTriggerPtr(3), DitsInitMessBulk(3), DitsGetEntReason(3), DitsGetEntBulkInfo(3), DitsPutActions(3), DitsArgIsBulk(3), DitsBulkArgRelease(3), DitsDefineShared(3), DitsDefineSdsShared(3), DitsBulkArgInfo(3), DitsGetArgument(3), DitsGetEntStatus(3).

Support: Tony Farrell, `AAO`

C.160 DitsUfaceCtxEnable — Enable user interface context for the current task.

Function: Enable user interface context for the current task.

Description: The routine enables user interfaces to be written for Dits tasks by enabling a Task to initiate messages in other tasks without having received a Dits message itself.

This routine can only be called outside of an action routine e.g. in the main loop or in a response routine specified by it. It's purpose is to enable a subset of Dits routines allowing a user interface to initiate and respond to Dits messages.

After this routine is called, a special context, `DITS_CTX_UFACE` is current.

When transactions are initiated from `UFACE` context, resulting messages result in a call to the `ResponseRoutine` specified in the preceding call to `DitsUfaceCtxEnable`.

When the `ResponseRoutine` is called with a context of `DITS_CTX_UFACE`, DitsGetCode() will return the code supplied in the call to `DitsUfaceCtxEnable`. Note that the only valid request to DitsPutRequest is `DITS_REQ_EXIT`.

You may want to call `DitsUfaceCtxEnable` in the response routine to change the handler for messages initiated by the response routine.

`UFACE` context should be re-enabled after each call to `DitsMsgReceive`.

The user interface context cannot chose to ignore messages, you cannot call DitsNotInterested on their behalf. All messages resulting from the transaction will be delivered to the response routine.

Calls to MsgOut will cause the routine put by DitsUfacePutMsgOut to be invoked. If no such routine is supplied, the message will be written to the standard output device for the program.

Output from Ers routines will cause the routine put by DitsUfacePutErsOut to be invoked. If no such routine is supplied, the message will be written to the standard error output device for the program.

Note that not all routines which can normally be invoked from a action routine are sensible when running in uface context. See the individual routine descriptions for possible restrictions.

Language: C

Call:

(Void) = DitsUfaceCtxEnable(ResponseRoutine, code, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) ResponseRoutine (DitsActionRoutineType) The routine to be called. See Dits documentation for details.

(>) code (long int) Will be returned by DitsGetCode in the response routine.

(!) status (StatusType *) Modified status.

Function Prototypes Used: typedef `DVOID` (*DitsActionRoutineType)(StatusType *status);

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(!) status (StatusType *) Modified status. The handler function should set status bad when it fails.

Include files: DitsSys.h

External functions used: none

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called from a Dits main routine (Not a dits application routine) or from the response routine.

See Also: The Dits Specification Document, DitsUfacePutErsOut(3), DitsUfacePutMsgOut(3), DitsUfaceRespond(3), DitsUfaceTimer(3), DitsUfaceTimerCancel(3), DitsUfaceWaitComp(3), DitsUfaceWaitInit(3), DulExecuteCmd(3)

Support: Tony Farrell, `AAO`

C.161 DitsUfaceErsRep — Given a Ers message structure, report it using ErsRep.

Function: Given a Ers message structure, report it using ErsRep.

Description: This routine takes apart the structure `DRAMA` uses to send Ers messages and reports the message using ErsRep.

Language: C

Call:
(Void) = DitsUfaceErsRep(id,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) id (SdsIdType id) The `SDS` id of the structure.

(`!`) status (StatusType `*`) Modified status.

Include files: DitsSys.h

Prior requirements: Should only be used as a Dits User interface reponse routine.

See Also: The Dits Specification Document, DitsUfaceCtxEnable(3), DitsUfacePutErsOut(3), DitsUfacePutMsgOut(3), DitsUfaceTimer(3), DitsUfaceTimerCancel(3), DitsUfaceWaitComp(3), DitsUfaceWaitInit(3), DulExecuteCmd(3).

Support: Tony Farrell, `AAO`

C.162 DitsUfaceMsgOut — Given an Info message structure, report it using MsgOut.

Function: Given an Info message structure, report it using MsgOut.

Description: This routine takes apart the structure `DRAMA` uses to send MsgOut messages and reports the message using MsgOut.

Language: C

Call:
(Void) = DitsUfaceMsgOut(id,status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) id (SdsIdType id) The SDS id of the structure.

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

Prior requirements: Should only be used as a Dits User interface reponse routine.

Support: Tony Farrell, AAO

C.163 DitsUfacePutErsOut — Put the routine used to output error message during Uface context.

Function: Put the routine used to output error message during Uface context.

Description: This function changes the routine used by Ers to write to the user when Uface context is enabled. Normally, calls to Ers output routine under Uface context result in the message being written to Stderr.

Note, this function is implemented as a macro.

Language: C

Call:
(Void) = DitsUfacePutErsOut (routine,client_data,status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) routine (ErsOutRoutineType) The new Ers output routine. See Ers documentation for details. If 0, revert to using stderr.

(>) client_data (Void *) Passed directly to the output routine as its outArg argument.

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used: None

External values used: DitsTask

Prior requirements: DitsAppInit should have been called.

See Also: The Dits Specification Document, DitsUfaceCtxEnable(3), DitsUfacePutMsgOut(3), DitsUfaceRespond(3), DitsUfaceTimer(3), DitsUfaceTimerCancel(3), DitsUfaceWaitComp(3), DitsUfaceWaitInit(3), DulExecuteCmd(3).

Support: Tony Farrell, AAO

C.164 DitsUfacePutMsgOut — Put the routine used for MsgOut output during Uface context.

Function: Put the routine used for MsgOut output during Uface context.

Description: This function changes the routine used by the MsgOut routine to write to the user when Uface context is enabled. Normally, calls to MsgOut under Uface context result in the message being written to Stdout.

Note, this function is implemented as a macro.

Language: C

Call:
(Void) = DitsUfacePutMsgOut (routine,client_data,status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) routine (DitsMsgOutRoutine) The new MsgOut output routine. If 0, revert to using stdout.

(>) client_data (Void *) Passed directly to the output routine.

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used: None

External values used: DitsTask

Prior requirements: DitsAppInit(3) should have been called.

See Also: The Dits Specification Document, DitsUfaceCtxEnable(3), DitsUfacePutErsOut(3), DitsUfaceRespond(3), DitsUfaceTimer(3), DitsUfaceTimerCancel(3), DitsUfaceWaitComp(3), DitsUfaceWaitInit(3), DulExecuteCmd(3).

Support: Tony Farrell, AAO

C.165 DitsUfaceRespond — General Uface context response routine.

Function: General Uface context response routine.

Description: This routine uses ErsOut and MsgOut to output messages to the user describing the reason for the entry. For most messages, a simple string is output giving the reason and associated status for an entry.

For Informational and Error messages received from a subsidiary task (messages sent by ErsOut/ErsFlush and MsgOut), the message is decoded and output using ErsRep and MsgOut as appropriate.

This routine can be specified to DitsUfaceCtxEnabled or called from whatever routine is supplied to DitsUfaceCtxEnabled. In the later case, is it acceptable that this routine only be called to respond to certain entries (such as MsgOut and ErsOut messages which require complex Sds translation).

Language: C

Call:
(Void) = DitsUfaceRespond(status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used:

ErsOut	Ers	Output an error message
DitsErrorText	Dits	Get the text associated with an error code.
DitsGetEntInfo	Dits	Get entry details
DitsGetReason	Dits	Get reason for entry
DitsPrintReason	Dits	Print the reason for an entry
ArgFind	Sds	Find an Sds item by name
SdsInfo	Sds	Get information on an Sds item
SdsGet	Sds	Get the contents of an Sds item

External values used: DitsTask - Details of the current task

Prior requirements: Should only be used as a Dits User interface reponse routine.

See Also: The Dits Specification Document, DitsUfaceCtxEnable(3), DitsUfacePutErsOut(3), DitsUfacePutMsgOut(3), DitsUfaceTimer(3), DitsUfaceTimerCancel(3), DitsUfaceWaitComp(3), DitsUfaceWaitInit(3), DulExecuteCmd(3).

Support: Tony Farrell, AAO

C.166 DitsUfaceTimer — Set up Uface Context timer message.

Function: Set up Uface Context timer message.

Description: Routines running in Uface Context cannot setup timer based reschedules using DitsPutRequest/DitsPutDelay as they are not running in the context of an action. This routine allows uface routines to explictly setup timers which result in a call to the specified response routine.

Additionally, application routines can invoke this routine to setup timers which will be triggered in Uface context.

This routine works by creating a message which looks like a message from another task.

Note that the response routine is invoked in uface context, but you should call DitsUfaceCtxEnable to install a new response routine before initiating messages to other tasks. Alternately, you can signal an action using DitsSignal().

In the ResponseRoutine, DitsGetReason will return a type of `DITS_REA_RESCHED` and DitsGetEntInfo will return the transaction id returned by this routine.

Before the timer has gone off, the transation id can be supplied to DitsUfaceTimerCancel to cancel the timer.

Language: C

Call:

(Void) = DitsUfaceTimer(delay, ResponseRoutine, code, transid, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) delay (DitsDeltaTimeType *) The delay before the timeout occurs. If a null pointer is supplied, a zero second timer is used.

(>) ResponseRoutine (DitsActionRoutineType) The routine to be called. See Dits documentation for details. If zero, the this value and the code are taken from the current Uface Context Response Routine as set by DitsUfaceCtxEnable. If not in Uface context, then this routine Must be supplied.

(>) code (long int) Will be returned by DitsGetCode in the response routine. Only used if Response Routine is supplied, otherwise the current value supplied to DitsUfaceCtxEnable is used.

(<) transid (DitsTransIdType *) Transaction id for the transaction started. If a null pointer is supplied, no transaction id is returned.

(!) status (StatusType *) Modified status.

Function Prototypes Used: typedef `DVOID` (*DitsActionRoutineType)(StatusType *status);

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(!) status (StatusType *) Modified status. The handler function should set status bad when it fails

Include files: DitsSys.h

External functions used:

Dits___TransIdCreate	Dits Internal	Create a transaction id
Dits___TransIdDelete	Dits Internal	Delete a transaction id
ImpQueueReminder	Imp	Queue a reminder message

External values used: DitsTask - Details of the current task

Prior requirements: DitsAppInit must have been called.

See Also: The Dits Specification Document, DitsUfaceCtxEnable(3), DitsUfaceTimerCancel(3), DulExecuteCmd(3).

Support: Tony Farrell, AAO

C.167 DitsUfaceTimerCancel — Cancel a timer messsage requested by DitsUfaceTimer.

Function: Cancel a timer messsage requested by DitsUfaceTimer.

Description: A timer specified by transid is deleted.

Language: C

Call:
(Void) = DitsUfaceTimerCancel(transid, status)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(>) transid (DitsTransIdType) Transaction id for the transaction as returned by DitsUfaceTimer.

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used:

ImpClearReminder	Imp	Clear a reminder
Dits___TransIdDelete	Dits internal	Delete a transaction id.

External values used: DitsTask - Details of the current task

Prior requirements: DitsAppInit must have been called.

See Also: The Dits Specification Document, DitsUfaceCtxEnable(3), DitsUfaceTimer(3), DulExecuteCmd(3).

Support: Tony Farrell, AAO

C.168 DitsUfaceTransIdWait — Blocks the current UFACE context and waits for a message to be received.

Function: Blocks the current UFACE context and waits for a message to be received.

Description: The current UFACE context is blocked and a message receive loop entered. Other actions and UFACE messages can be processed as normal. When a message concerning this context is received, the context is unblocked and will continue. The current entry details (DitsGetArgument() DitsGetEntInfo() etc) will then be for the first message received for this UFACE context. The count variable will return the number of other messages still remaining to be processed.

NOTE - the argument retrieved by DitsGetArgument() will be an SdsCopy(3) of the original argument structure. As a result, you should not use DitsUfaceWait(3) to handle messages expecting replies with large argument structures.

In order to associate messages with this context, you must call DitsUfaceWaitInit() before calling the message sending routines (DitsGetPath(), DitsObey() etc.) DitsUfaceWaitInit() replaces a call to DitsUfaceCtxEnable() and you can revert to that style by calling DitsUfaceCtxEnable(). When you have finished processing messages, you should call DitsUfaceWaitComp(). You can have mulitple calls to DitsUfaceWait() between DitsUfaceWaitInit() and DitsUfaceWaitComp().

If the transaction id argument is non-zero, then will only return if that transaction completes.

The outstanding messages can be processed by calling this routine repeatly. The call will block if the last call. returned a count of 0, except if the DONT_BLOCK flag is set.

Note that if DitsUfaceWait/DitsUfaceTransIdWait/DitstActionWait/ DitsActionTransIdWait are called by other actions or events invoked whilst this call is active, the second action/UFACE context must be unblocked and end/reschedule before control will return to the first call.

If messages received while the context was blocked have not been processed when the DitsUfaceWaitComp is invoked, they are lost (a message is output using ErsOut). To ensure these messages are handled, you should continue calling this routine until count is returned as 0.

You can freely mix calls to DitsUfaceWait() and DitsUfaceTransIdWait().

Language: C

Call:

(void) = DitsUfaceTransIdWait(flags,timeout,info, transId, count,status);

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

(>) flags (int) A bit mask of flags. The possibilities are any or-ed combination of-

DITS_M_AW_FORGET	Forget any remaining messages from previous calls to this routine for the the info block.
DITS_M_AW_DONT_BLOCK	Always return immediately. We will only read a message if there was one outstanding from a previous call. If there are no outstanding messages or DITS_M_FORGET flag has been set, then no message will be read and count will be set to -1. Action

() details (such as DitsGetArgument() etc.) will then be as per prior to the call.

DITS_M_AW_NO_LIST	Don't take an entry from the list. If combined with DONT_BLOCK we simply get the count of outstanding messages. Otherwise we block waiting for a message. When the call returns, it will be as per the first item on the list. It is not clear if this flag is usefull to users without the DONT_BLOCK flag.

(>) timeout (DitsDeltaTimeType *) If non-zero, the address of a reschedule delay.

(>) info (DitsUfaceWaitInfoType *) A variable initialised by calling DitsUfaceWaitInit prior to a sequence of calls to this routine. It is used to maintain details about this context.

(>) transId (DitsTransIdType) IF non-zero, the ID of a transaction to wait for. This must be a transaction ID of a message sent (or to be delivered to) this action.

(<) count (int *) Set to the number of messages remaining to be processed. The message which caused the routine to be unblocked is not included. If the DITS_M_DONT_BLOCK flag was set, then count will be -1 if no messages were available. Count is always no-negative in other cases.

(!) status (StatusType *) Modified status. WARNING - status of message receiving code, not message status. To get the status of the received message, fetch the value from DitsGetEntStatus().

Include files: DitsSys.h, DitsInteraction.h

External functions used:

ImpReadEnd	Imp	Indicate completion of a read by pointer
SdsDelete	Sds	Delete a structure
SdsFreeId	Sds	Free a id
DitsMsgReceive	Dits	Received a dits message.
DitsUfaceTimer	Dits	Create a UFACE context timer.
DitsUfaceTimerCancel	Dits	Cancel a UFACE context timer.

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called at `UFACE` context.

See Also: The Dits Specification Document, DitsUfaceCtxEnable(3), DitsActionWait(3), DitsUfaceWaitComp(3), DitsUfaceWaitInit(3) .

Support: Tony Farrell, `AAO`

C.169 DitsUfaceWait — Blocks the current `UFACE` context and waits for a message to be received.

Function: Blocks the current `UFACE` context and waits for a message to be received.

Description: The current `UFACE` context is blocked and a message receive loop entered. Other actions and `UFACE` messages can be processed as normal. When a message concerning this context is received, the context is unblocked and will continue. The current entry details (DitsGetArgument() DitsGetEntInfo() etc) will then be for the first message received for this `UFACE` context. The count variable will return the number of other messages still remaining to be processed.

`NOTE` - the argument retrieved by DitsGetArgument() will be an SdsCopy(3) of the original argument structure. As a result, you should not use DitsUfaceWait(3) to handle messages expecting replies with large argument structures.

In order to associate messages with this context, you must call DitsUfaceWaitInit() before calling the message sending routines (DitsGetPath(), DitsObey() etc.) DitsUfaceWaitInit() replaces a call to DitsUfaceCtxEnable() and you can revert to that style by calling DitsUfaceCtxEnable(). When you have finished processing messages, you should call DitsUfaceWaitComp(). You can have mulitple calls to DitsUfaceWait() between DitsUfaceWaitInit() and DitsUfaceWaitComp().

The outstanding messages can be processed by calling this routine repeatly. The call will block if the last call. returned a count of 0, except if the `DONT_BLOCK` flag is set.

Note that if DitsUfaceWait/DitsUfaceTransIdWait/DitstActionWait/ DitsActionTransIdWait are called by other actions or events invoked whilst this call is active, the second action/`UFACE` context must be unblocked and end/reschedule before control will return to the first call.

If messages received while the context was blocked have not been processed when the DitsUfaceWaitComp is invoked, they are lost (a message is output using ErsOut). To ensure these messages are handled, you should continue calling this routine until count is returned as 0.

You can freely mix calls to DitsUfaceWait() and DitsUfaceTransIdWait().

Language: C

Call:

(void) = DitsUfaceWait(flags,timeout,info,count,status);

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) flags (int) A bit mask of flags. The possiblities are any or-ed combination of-

`DITS_M_AW_FORGET`	Forget any remaining messages from previous calls to this routine for the the info block.
`DITS_M_AW_DONT_BLOCK`	Always return immediatley. We will only read a message if there was one outstanding from a previous call. If there are no outstanding messages or `DITS_M_FORGET` flag has been set, then no message will be read and count will be set to -1. Action

() details (such as DitsGetArgument() etc.) will then be as per prior to the call.

`DITS_M_AW_NO_LIST`	Don't take an entry from the list. If combined with `DONT_BLOCK` we simply get the count of outstanding messages. Otherwise we block waiting for a message. When the call returns, it will be as per the first item on the list. It is not clear if this flag is usefull to users without the `DONT_BLOCK` flag.

(>) timeout (DitsDeltaTimeType *) If non-zero, the address of a reschedule delay.

(>) info (DitsUfaceWaitInfoType *) A variable initialised by calling DitsUfaceWaitInit prior to a sequence of calls to this routine. It is used to maintain details about this context.

(<) count (int *) Set to the number of messages remaining to be processed. The message which caused the routine to be unblocked is not included. If the `DITS_M_DONT_BLOCK` flag was set, then count will be -1 if no messages were available. Count is always no-negative in other cases.

(!) status (StatusType *) Modified status. `WARNING` - status of message receiving code, not message status. To get the status of the received message, fetch the value from DitsGetEntStatus().

Include files: DitsSys.h, DitsInteraction.h

External functions used:

ImpReadEnd	Imp	Indicate completion of a read by pointer
SdsDelete	Sds	Delete a structure
SdsFreeId	Sds	Free a id
DitsMsgReceive	Dits	Received a dits message.
DitsUfaceTimer	Dits	Create a `UFACE` context timer.
DitsUfaceTimerCancel	Dits	Cancel a `UFACE` context timer.

External values used: DitsTask - Details of the current task

Prior requirements: Can only be called at `UFACE` context.

See Also: The Dits Specification Document, DitsUfaceCtxEnable(3), DitsActionWait(3), DitsUfaceWaitComp(3), DitsUfaceWaitInit(3) .

Support: Tony Farrell, `AAO`

C.170 DitsUfaceWaitComp — Tidy up after calls to DitsUfaceWait.

Function: Tidy up after calls to DitsUfaceWait.

Description: There should be a matching call to this routine for each call to DitsUfaceInit(3). It tidys up outstanding message details and ensures the task internals are correct.

You can have multiple calls to DitsUfaceWait(3) between calls to DitsUfaceWaitInit(3) and DitsUfaceWaitComp(3).

Language: C

Call:
(void) = DitsUfaceWaitComp (info,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) info (DitsUfaceWaitInfoType `*`) A variable initialised by calling DitsUfaceWaitInit().

(`!`) status (StatusType `*`) Modified status.

Include files: DitsSys.h, DitsInteraction.h

External functions used:

free	`CRTL`	Free memory.

External values used: DitsTask - Details of the current task

Prior requirements: none

See Also: The Dits Specification Document, DitsUfaceCtxEnable(3), DitsActionWait(3), DitsUfaceWait(3), DitsUfaceWaitInit(3) .

Support: Tony Farrell, AAO

C.171 DitsUfaceWaitInit — Setup for a call to DitsUfaceWait.

Function: Setup for a call to DitsUfaceWait.

Description: This routine should be invoked prior to any call which will send a message (such as DitsObey), when it is intended to use DitsUfaceWait to wait for a response from the procedure.

There should be a matching call to DitsUfaceWaitComp. Between the two calls you can use DitsUfaceWait as many times as necessary.

Language: C

Call:
(void) = DitsUfaceWaitInit (info, handleInfo,status)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(>) info (DitsUfaceWaitInfoType *) Used internally by DitsUfaceWait. It should be passed to the matching call to DitsUfaceWaitComp.

(>) handleInfo (int) If set true, the system will automatically forward ERROR and MsgOut messages to the user interface.

(!) status (StatusType *) Modified status.

Include files: DitsSys.h

External functions used:

External values used: DitsTask - Details of the current task

Prior requirements: DitsAppInit(3) must have been called.

Support: Tony Farrell, AAO

C.172 MsgOut — Sends a message to the user interface.

Function: Sends a message to the user interface.

Description: This routine creates a `DITS_MSG_MESSAGE` message and sends it to the initiator of the current action. Normally, such a message will be forwarded to the user interface.

The format and it's arguments are the the same as used by the `printf` C `RTL` function.

The initiator of the current action may choose to handle the message itself or to forward it to its own initiator (See DitsInterested). In any case, at some level, the message must be handled and if appropriate, output to the user interface.

When an initiating action selects to handle such messages, it will be invoked with a reason code of `DITS_REA_MESSAGE`. The action can obtain the details of the message by fetching its argument using DitsGetArgument. This will return an Sds id of an item with two components. The first component is a one dimensional character string named `TASKNAME`, which contains the task name of the originator of the message. The second component is a two dimensional character array, named `MESSAGE`. This component is an array of the message texts. (currently, the second dimension is always 1).

When this routine is called by a routine with the context `DITS_CTX_UFACE`, there is no originator to send the message to. Such message will be output using the routine will be output using the routine specified with DitsUfacePutMsgOut or simply be output to stdout in the format "taskname:message" if no such routine has been put.

Language: C

Call:

(Void) = MsgOut (status, format, [arg ,[...]])

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`!`) status (StatusType `*`) Modified status.

(`>`) format (Char `*`) A format statement. See the description of the C printf function.

(`>`) arg (assorted) Formating arguments. Set the description of the C printf function.

Include files: DitsMsgOut.h

External functions used:

va_start	Crtl	Start a variable argument session.
va_args	Crtl	Get an argument.
va_end	Crtl	End a variable argument session.
vprintf	Crtl	Formated print (with variable arg list).
printf	Crtl	Formated print.
Dits___SendTap	Imp internal	Send a message to the originator of an action.
SdsNew	Sds	Create a Sds item.
SdsPut	Sds	Put data into an sds item.
SdsDelete	Sds	Delete an Sds item.
SdsFreeId	Sds	Free an Sds id.
ErsVSPrintf	Ers	Formated print to a string (with variable arg list).

External values used: DitsTask - details of the current task.

Prior requirements: Can only be called from a Dits application routine or a user interface response routine.

See Also: The Dits Specification Document, DitsUfacePutMsgOut(3), Ers specification.

Support: Tony Farrell, `AAO`

D Sdp routines

These routines implement the Simple Dits parameter system. They can be used in conjunction with the `ARG` routines supplied with `SDS`.

D.1 SdpCreate — Create some parameters.

Function: Create some parameters.

Description: The supplied Sdp parameter definition array gives details of parameters which are to be created or modified.

If a parameter of the given name does not exist, then it is created using the specified type. If is does exist and its type is not compatible with the new type, then it is destroyed and recreated in the new type. In both cases, it is given the supplied value.

The allowed types are-

`SDP_CHAR`	A single Character
`SDP_BYTE`	A signed Byte
`SDP_UBYTE`	An unsigned Byte
`SDP_SHORT`	A signed short
`SDP_USHORT`	An unsigned short
`SDP_INT`	A signed 32 bit integer
`SDP_UINT`	A unsigned 32 bit integer
`SDP_INT64`	A signed 64 bit integer
`SDP_UINT64`	A unsigned 64 bit integer
`SDP_FLOAT`	A float
`SDP_DOUBLE`	A double length float
`SDP_STRING`	A character string.
`SDSP_SDS`	The value is the Id of an `SDS` Item The item is renamed to the name of the parameter and inserted directly into the parameter system. Such an item can be any Sds item.

The value element should be the address of the value. If normal length of a value of that time is assumed. If the value is a string, then it is assumed to be null terminated.

Note, for types `SDP_INT/SDP_UINT`, you must use the types `INT32` and `UINT32` to contain the corresponding default values to ensure portability.

Language: C

Call:
(Void) = SdpCreate (id, size, pardefs, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) id (`VOIDP`) Id of the Sdp structure as returned by SdpInit.

(>) size (Long Int) The number of parameter in pardef

(>) pardefs (SdpParDefType`[]`) Deatils of parameters to be creates. This is an array with each element of the form-
{ char * name, void * value, int type }

where name is a null terminated string and type is one of the types listed above.

(`!`) status (StatusType `*`) Modified status.

Include files: Sdp.h

External functions used:

SdsFind	Sds	Find an Sds item by name
SdsInfo	Sds	Find details of an Sds item
SdsDelete	Sds	Delete an sds Item.
SdsNew	Sds	Create an Sds item
SdsPut	Sds	Put the value of an Sds item.
SdsRename	Sds	Rename an Sds item.
strlen	Crtl	Return the length of a string.

External values used: none

Prior requirements: SdpInit must have been called.

See Also: The Dits Specification Document, SdpInit(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3),

Support: Tony Farrell, `AAO`

D.2 SdpCreateItem — Create a single parameter from an Sds item.

Function: Create a single parameter from an Sds item.

Description: The supplied Sds item is inserted into the specified parameter system. Any existing item of the same name is deleted.

Language: C

Call:
(Void) = SdpCreateItem (parsys, item, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) parsys (SdsIdType) Id of the Sdp structure as returned by SdpInit.

(>) item (SdsIdType) The Sds item to be inserted. This item will be inserted into the Sdp structure using SdsInsert. Must not be an external item.

(!) status (StatusType *) Modified status.

Include files: Sdp.h

External functions used:

SdsFind	Sds	Find an Sds item by name
SdsInfo	Sds	Find details of an Sds item
SdsDelete	Sds	Delete an sds Item.
SdsInsert	Sds	Inset an sds item.

External values used: none

Prior requirements: The Sdp Id should have been created by SdpInit.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpGetx(3), SdpPutx(3),

Support: Tony Farrell, `AAO`

D.3 SdpGet — Return the value of a parameter of a given name.

Function: Return the value of a parameter of a given name.

Description: The routine is intended to be specified to DitsAppParamSys as the getRoutine argument.

It finds the parameter by name within the parameter system specified by id. It creates a new Sds structure into which it makes a copy of the required parameter. The structure has a name of ArgStructure and is suitable for access by the Arg routines.

This routine uses the SdsFindByPath routine to find the item and hence can be used to access subsidary parts of a structure.

Language: C

Call:
(Void) = SdpGet (id, name, parid, delete, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) id (void *) An SdpId as returned by SdpInit.

(>) name (Char *) The null terminated name of the parameter to return. The following special names are supported

`_ALL_`	This requests that the entire parameter system be returned. In this case, an SdpStructure item is returned.
`_NAMES_`	Requests that a list of names be returned. In this case a structure of type SdpNames is returned. This will contain one item, an array with each of the names in it.

(<) parid (SdsIdType *) The Sds id of the parameter

(<) delete (int *) Set true to indicate parid sds item should be deleted when Dits is finished with it.

(!) status (StatusType *) Modified status.

Include files: Sdp.h

External functions used:

SdsNew	Sds	Create a new Sds item.
SdsFindByPath	Sds	Find an Sds item.
SdsCopy	Sds	Make a copy of an Sds item.
SdsInsert	Sds	Insert one sds item into another.

External values used: none

Prior requirements: SdpInit must have been called.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3) SdpSet(3), SdpMGet(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell, `AAO`

D.4 SdpGetSds — Get a Sds item from a parameter.

Function: Get a Sds item from a parameter.

Description: A sds item is read from a named parameter. This routine returns an Sds Id which refers to the actual parameter, within the parameter system. Thus modification via the returned id will modify the actual parameter item. You can use SdpUpdate to notify the parameter system of such changes (which is required to ensure that other tasks which monitor this parameter get notification of changes)

Note that a new Sds id is allocated and should be free-ed when you are finished with it by calling SdsFreeId. You should not delete this Sds item as doing so will delete the actual parameter.

Language: C

Declaration: void SdpGetSds(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) name (char*) Name of the component to be read from.

(<) value (SdsIdType *) Sds id of the result. Should be free-ed when you are finished with it.

(!) status (StatusType*) Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3), Sds manual.

Support: Tony Farrell, `AAO`

D.5 SdpGetString — Get a character string item from a parameter

Function: Get a character string item from a parameter

Description: A character string item is read from a named parameter.

Language: C

Declaration: void SdpGetString(name, len, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) name (char*) Name of the component to be read from.

(>) len (long) Length of buffer to receive string.

(<) value (char*) Character string buffer to read into.

(!) status (StatusType*) Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell, `AAO`

D.6 SdpGetSys — Return the `SDS ID` of the entire parameter system.

Function: Return the `SDS ID` of the entire parameter system.

Description: Returns the `SDS ID` of the entire parameter system. If zero is returned, we don't have one.

Language: C

Declaration: void SdpGetSys()

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3), Sds manual.

Support: Tony Farrell, `AAO`

D.7 SdpGetc — Get a character item from a parameter.

Function: Get a character item from a parameter.

Description: A character item is read from a named parameter.

Language: C

Declaration: void SdpGetc(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) name (char*) Name of the component to be read from.

(>) value (char*) Character variable to read into.

(!) status (StatusType *) Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell, `AAO`

D.8 SdpGetd — Get a double floating point item from a parameter.

Function: Get a double floating point item from a parameter.

Description: A double floating point item is read from a named parameter,

Language: C

Declaration: void SdpGetd(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(`>`) name (char*)** Name of the component to be read from.
- **(`>`) value (double*)** Double variable to read into.
- **(`!`) status (StatusType `*`)** Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell, `AAO`

D.9 SdpGetf — Get a floating point item from a parameter.

Function: Get a floating point item from a parameter.

Description: A floating point item is read from a named parameter.

Language: C

Declaration: void SdpGetf(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(`>`) name (char*)** Name of the component to be read from.
- **(`>`) value (float*)** Float variable to read into.
- **(`!`) status (StatusType `*`)** Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell, `AAO`

D.10 SdpGeti — Get a long integer item from a parameter

Function: Get a long integer item from a parameter

Description: A long integer integer item is read from a named parameter.

Language: C

Declaration: void SdpGeti(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(>) name (char*)** Name of the component to be read from.
- **(>) value (long*)** Long variable to read into.
- **(!) status (StatusType *)** Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell, `AAO`

D.11 SdpGeti64 — Get a 64 bit integer item from a parameter

Function: Get a 64 bit integer item from a parameter

Description: A 64 bit integer integer item is read from a named parameter.

Language: C

Declaration: void SdpGeti64(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(>) name (char*)** Name of the component to be read from.
- **(>) value (`INT64`*)** Long variable to read into.
- **(!) status (StatusType *)** Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell, `AAO`

D.12 SdpGets — Get a short integer item from a parameter.

Function: Get a short integer item from a parameter.

Description: A short integer item is read from a named parameter.

Language: C

Declaration: void SdpGets(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(>) name (char*)** Name of the component to be read from.
- **(>) value (short*)** Short variable to read into.
- **(!) status (StatusType *)** Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell.

D.13 SdpGetu — Get an unsigned integer item from a parameter.

Function: Get an unsigned integer item from a parameter.

Description: An unsigned long integer item is read from a named parameter.

Language: C

Declaration: void SdpGetu(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(>) name (char*)** Name of the component to be read from.
- **(>) value (`UINT64*`)** Write the value here
- **(!) status (StatusType *)** Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell, `AAO`

D.14 SdpGetu64 — Get a 64 bit unsigned integer item from a parameter.

Function: Get a 64 bit unsigned integer item from a parameter.

Description: A unsigned 64 bit integer item is read from a named parameter.

Language: C

Declaration: void SdpGetu64(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(`>`) name (char*)** Name of the component to be read from.
- **(`>`) value (`UINT64*`)** Write the value here
- **(`!`) status (StatusType `*`)** Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell, `AAO`

D.15 SdpGetus — Get an unsigned short integer item from a parameter.

Function: Get an unsigned short integer item from a parameter.

Description: An unsigned short integer item is read from a named parameter.

Language: C

Declaration: void SdpGetus(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(`>`) name (char*)** Name of the component to be read from.
- **(`>`) value (unsigned short*)** Short variable to read into.
- **(`!`) status (StatusType `*`)** Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell, `AAO`

D.16 SdpInit — Initialise the Simple Dits parameter system.

Function: Initialise the Simple Dits parameter system.

Description: Initialise the A Dits parameter system using the Sdp parameter structure. We create the Sds structure and call DitsAppParamSys We return the id for us in calls to SdpCreate.

Language: C

Call:
(Void) = SdpInit (id, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(<) id (SdsIdType *) Sds Identifier to the created structure

(!) status (StatusType *) Modified status.

Include files: Sdp.h

External functions used:

SdsNew	Sds	Create a new sds structure.
DitsAppParamSys	Dits	Add the parameter system to Dits.

External values used: none

Prior requirements: DitsAppInit must have been called.

See Also: The Dits Specification Document, SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3), DitsAppInit(3), DitsAppParamSys(3).

Support: Tony Farrell, `AAO`

D.17 SdpMGet — Return the values of a list of parameters.

Function: Return the values of a list of parameters.

Description: The routine is intended to be specified to DitsAppParamSys as the mGetRoutine argument.

It breaks the list of names up and finds each parameter by name within the parameter system specified by id. It creates a new Sds structure into which it makes a copy of the required parameter. The structure has a name of ArgStructure and is suitable for access by the Arg routines.

Language: C

Call:

(Void) = SdpMGet (id, name, parid, status)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(>) id (SdsIdType) An SdpId as returned by SdpInit.

(>) names (Char *) The null terminated list of names of the parameter to return. The following special names are supported

ALL	This requests that the entire parameter system be returned. In this case, an SdpStructure item is returned.
NAMES	Requests that a list of names be returned. In this case a structure of type SdpNames is returned. This will contain one item, an array with each of the names in it.

(<) parid (SdsIdType *) The Sds id of the parameter

(<) delete (int *) Set true to indicate parid sds item should be deleted when Dits is finished with it.

(!) status (StatusType *) Modified status.

Include files: Sdp.h

External functions used:

SdsNew	Sds	Create a new Sds item.
SdsFind	Sds	Find an Sds item.
SdsCopy	Sds	Make a copy of an Sds item.
SdsInsert	Sds	Insert one sds item into another.

External values used: none

Prior requirements: SdpInit must have been invoked.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpGet(3), sdpSet(3)

Support: Tony Farrell, AAO

D.18 SdpPutSds — Put a sds item into a parameter

Function: Put a sds item into a parameter

Description: An sds item is written into a named parameter. Conversion is carried out using ArgCvt. This means that if both input and output items are scaler or strings, they will be converted if possible. Otherwise, a conversion error will occur.

To put the value of a structured Sds item, use SdpGetSds to get an id for it and then write the value using that id. Follow your update by a call to SdpUpdate. Alternatively, use SdsPutStruct which replaces the old structure with a new one.

Language: C

Declaration: void SdpPutSds(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) name (char*) Name of the component to be written to.

(>) value (SdsIdType) Sds item to be written.

(!) status (StatusType *) Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3). Sds manual.

Support: Tony Farrell, `AAO`

D.19 SdpPutString — Put a character string item into a parameter

Function: Put a character string item into a parameter

Description: A character string item is written into a named parameter

Language: C

Declaration: void SdpPutString(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) name (char*) Name of the component to be written to.

(>) value (char*) Null terminated string to be written.

(!) status (StatusType *) Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell, `AAO`

D.20 SdpPutStruct — Put a sds structure into a parameter.

Function: Put a sds structure into a parameter.

Description: An sds item is written into a named parameter which must be a structured item. The named parameter's value is replaced by this structure.

Language: C

Declaration: void SdpPutStruct(name, value, copy, create, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(>) name (char*)** Name of the component to be written to.
- **(>) value (SdsIdType)** Sds structure item to be written.
- **(>) copy (int)** If true, we must insert a copy of the Sds structure refered to by value into the parameter system. If false, we can use that item directly.
- **(>) create (int)** If true and parameter does not yet exist, create it. This flag is normally used when initally setting up a `ARG_SDS` type parameter whose value can't be initialised using SdpCreate.
- **(!) status (StatusType *)** Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3). Sds manual.

Support: Tony Farrell, `AAO`

D.21 SdpPutc — Put a character item into a parameter

Function: Put a character item into a parameter

Description: A character item is written into a named parameter.

Language: C

Declaration: void SdpPutc(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(>) name (char*)** Name of the component to be written to.
- **(>) value (char)** Character value to be written.
- **(!) status (StatusType *)** Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell, `AAO`

D.22 SdpPutd — Put a double floating point item into a parameter.

Function: Put a double floating point item into a parameter.

Description: A double item is written into a named parameter.

Language: C

Declaration: void SdpPutd(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(>) name (char*)** Name of the component to be written to.
- **(>) value (double)** Floating point value to be written.
- **(!) status (StatusType *)** Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell, `AAO`

D.23 SdpPutf — Put a floating point item into a parameter

Function: Put a floating point item into a parameter

Description: A floating point item is written into a named parameter

Language: C

Declaration: void SdpPutf(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(>) name (char*)** Name of the component to be written to.
- **(>) value (float)** Floating point value to be written.
- **(!) status (StatusType *)** Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell, `AAO`

D.24 SdpPuti — Put a integer item into a parameter.

Function: Put a integer item into a parameter.

Description: A long integer item is written into a named parameter.

Language: C

Declaration: void SdpPuti(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(>) name (char*)** Name of the component to be written to.
- **(>) value (long)** Long integer value to be written.
- **(!) status (StatusType *)** Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell, `AAO`

D.25 SdpPuti64 — Put a 64 bit integer item into a parameter.

Function: Put a 64 bit integer item into a parameter.

Description: A 64 bit integer item is written into a named parameter.

Language: C

Declaration: void SdpPuti64(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(>) name (char*)** Name of the component to be written to.
- **(>) value (`INT64`)** Long integer value to be written.
- **(!) status (StatusType *)** Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell, `AAO`

D.26 SdpPuts — Put a short integer item into a parameter

Function: Put a short integer item into a parameter

Description: A short integer item is written into a named parameter.

Language: C

Declaration: void SdpPuts(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(>) name (char*)** Name of the component to be written to.
- **(>) value (short)** Short integer value to be written.
- **(!) status (StatusType *)** Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell, `AAO`

D.27 SdpPutu — Put an unsigned integer item into a parameter.

Function: Put an unsigned integer item into a parameter.

Description: An unsigned long integer item is written into a named parameter.

Language: C

Declaration: void SdpPutu(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(>) name (char*)** Name of the component to be written to.
- **(>) value (unsigned long)** Unsigned integer value to be written.
- **(!) status (StatusType *)** Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell, `AAO`

D.28 SdpPutu64 — Put a 64 bit unsigned integer item into a parameter.

Function: Put a 64 bit unsigned integer item into a parameter.

Description: An unsigned 64 bit integer item is written into a named parameter.

Language: C

Declaration: void SdpPutu64(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(>) name (char*)** Name of the component to be written to.
- **(>) value (`UINT64`)** Unsigned integer value to be written.
- **(!) status (StatusType *)** Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell, `AAO`

D.29 SdpPutus — Put an unsigned short integer item into a parameter.

Function: Put an unsigned short integer item into a parameter.

Description: An unsigned short item is written into a named parameter.

Language: C

Declaration: void SdpPutus(name, value, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(>) name (char*)** Name of the component to be written to.
- **(>) value (unsigned short)** Unsigned short value to be written.
- **(!) status (StatusType *)** Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3).

Support: Tony Farrell, `AAO`

D.30 SdpSet — Given an Sdp id, set the value of the given parameter.

Function: Given an Sdp id, set the value of the given parameter.

Description: The routine is intendend to be specified to DitsAppParamSys as the SetRoutine argument. It finds the parameter of the given name and sets the value of that parameter.

Any necessary conversions are performed if possible. `ARG__CNVERR` is returned if a conversion cannot be preformed.

parid is assumed to be an argument structure containing the item or the id of the item itself. If is is a structure containing the item, the first item in the structure is used.

The possible conversions are-

BasicType	Basic type (range violations possible)
BasicType	String (always works, except for length violation)
BasicType	Other type (works but see notes below)
String	Other type (works but see notes below)
String	Basic type (works if string can be represented)
String	String (always works, except for length violation)
OtherType	Other type (always works)

"Other type" means an Sds structured type or an array, other then an array of characters with only one dimensions (which is a string).

The impossible conversions are-

OtherType	String
OtherType	Basic type

Conversions to "Other type" are done by deleting the original parameter of the given name and inserting a copy of the "OtherType" value, which has been renamed to the name of a parameter.

Strings should be null terminated.

This routine uses the SdsFindByPath routine to find the item and hence can be used to access subsidary parts of a structure.

Language: C

Call:

(Void) = SdpSet (id, name, parid, status)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(>) id (void *) An SdpId as returned by SdpInit.

(>) name (Char *) The null terminated name of the parameter to set

(>) parid (SdsIdType *) The Sds id of the new value of the parameter.

(!) status (StatusType *) Modified status.

Include files: Sdp.h

External functions used:

SdsFindByPath	Sds	Find an Sds item
SdsInfo	Sds	Retrieve information about an item
SdsIndex	Sds	Index an Sds item.
SdsDelete	Sds	Delete a sds item.
SdsFreeId	Sds	Free a Sds id.
SdsCopy	Sds	Copy one Sds id to another.
SdsRename	Sds	Rename an Sds item.
SdsInsert	Sds	Insert an Sds item into structure.
ArgCvt	Arg	SDS type conversion
DitsMonitor	Dits	Notify dits monitor code of a change.

External values used: none

Prior requirements: SdpInit must have been called.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3) SdpGet(3), SdpGetx(3), SdpPutx(3), SdpSetReadonly(3).

Support: Tony Farrell, AAO

D.31 SdpSetReadonly — Set Sdp parameters to be readonly.

Function: Set Sdp parameters to be readonly.

Description: Set the Sdp parameter system such the parameters cannot be set from outside the task. Any such attempts will return a bad status.

Language: C

Call:
(Void) = SdpSetReadonly (status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

(!) status (StatusType *) Modified status.

Include files: Sdp.h

External functions used:

DitsAppParamSys	Dits	Add the parameter system to Dits.

External values used: none

Prior requirements: SdpInit must have been called.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3) SdpSet(3), SdpGetx(3), SdpPutx(3), SdpSet(3).

Support: Tony Farrell, `AAO`

D.32 SdpUpdate — Notify the parameter system that an item has been updated via its id.

Function: Notify the parameter system that an item has been updated via its id.

Description: The id is a value returned by SdpGetSds. It represents the Sds id of a parameter system item. This routine is used to notify the parameter system that the value of the parameter has been changed using the sds id.

Language: C

Declaration: void SdpUpdate(id, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) id (SdsIdType) A value returned from SdpGetSds.

(!) status (StatusType *) Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpGetx(3), SdpPutx(3), DitsMonitor(3).

Support: Tony Farrell, `AAO`

D.33 SdpUpdateByName — Notify the parameter system that an item has been updated.

Function: Notify the parameter system that an item has been updated.

Description: This routine is used to notify the parameter system that the value of a parameter has been updated by means other then via an Sdp routine. This might be as a result of SdpGetSds() or via a pointer to the value etc. Note, if you have the `ID` of the parameter, please use SdpUpdate() as that is more efficent, only use this function if you don't have the `ID` available.

Language: C

Declaration: void SdpUpdateByName(name, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) name (char*) Name of the parameter.

(!) status (StatusType *) Modified status.

See Also: The Dits Specification Document, SdpInit(3), SdpCreate(3), SdpCreateItem(3), SdpUpdate(3), SdpGetx(3), SdpPutx(3), DitsMonitor(3).

Support: Tony Farrell, `AAO`

E Dui routines

These routines are used to help build user interfaces.

E.1 DuiAnaMessBufSizes — Analyze a string describing message buffer sizes.

Function: Analyze a string describing message buffer sizes.

Description: Uses the DuiToken routines to analyze a string describing message buffer sizes.

The string may containg the following possibilities

```
int             Sets the Message Bytes value
int:int           MessageBytes:ReplyBytes
int:int:int       MessageBytes:ReplyBytes:MaxReplies
int:int:int:int   MessageBytes:MaxMessages:ReplyBytes:MaxReplies
```

Language: C

Call:

(Void) = DuiAnaMessBufSizes (string,MessageBytes,MaxMessages, ReplyBytes,MaxReplies,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) string (char *) The string containing the message buffer sizes description.

(>) MessageBytes (long int *) Message Bytes

(>) MaxMessags (long int *) Max Messages

(>) ReplyBytes (long int *) Reply Bytes

(>) MaxReplies (long int *) Max Replies

(!) status (StatusType *) Modified status

Include files: Dui.h

External functions used:

DuiTokenInit	Dui	Initialise a string token
DuiTokenNext	Dui	Return next token
DuiTokenShut	Dui	Release a string token
strtol	Crtl	Convert a string to an integer

External values used: none

Prior requirements: none

See Also: The Dits Specification Document, DuiTokenInit(3), DuiTokenNext(3).

Support: Tony Farrell, `AAO`

E.2 DuiCommand — Interperts and processes a command string

Function: Interperts and processes a command string

Description: The command string either contains the name of an action and its arguments or one of the following commands which should be prefixed by a backslash.

`OBEY`	Send an obey message. The rest of the line is the action and arguments to the action.
`KICK`	Send a kick message. The rest of the line is the action and arguments to the action.
`GET`	Send a Get Parameter message. The rest of the line is the parameter name and it's value.
`SET`	Send a Set Parameter message
`CONTROL`	Send a Control message. The rest of the line is the message name and it's arguments.
`MONITOR`	Send a Monitor message. The rest of the line is the message name and its arguments.
`UPPER`	Convert subsequent action names to upper case
`LOWER`	convert subsequent action names to lower case
`NOCHANGE`	Don't do a case conversion on subsequent action names.

The above commands can be abbreivated to one character and case is not significant (only in the above commands, not the rest of the line).

This routine is intended to be supplied to DuiInteractive as a command callback routine.

Language: C

Call:

(Void) = DuiCommand (CommandString, client_data, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) CommandString (char *) The command string

(>) client_data (void *) Actually should be a (DuiDetailsType *) with the buffer size and task name elements set. Also the UserData item of the DuiDetailsType should have one of the values

`DUI_C_UPPER`	Convert case of action names to upper case.
`DUI_C_LOWER`	Convert case of action names to lower case.
`DUI_C_NOCHANGE`	Convert convert case of action names. This is default after DuiDetailsInit is called.

(!) status (StatusType *) Modified status.

Include files: Dui.h

External functions used:

malloc	`CRTL`	Allocate memory
free	`CRTL`	Release allocated memory
strcmp	`CRTL`	Compare two strings.
strncmp	`CRTL`	Compare two strings.
strlen	`CRTL`	Return the length of a string.
strncpy	`CRTL`	Copy one string to another.
islower	`CRTL`	Is a characterlower case.
isupper	`CRTL`	Is a character upper case.
toupper	`CRTL`	Return the upper case version of a lower case char
tolower	`CRTL`	Return the lower case version of an upper case char
sprintf	`CRTL`	Formated print to a string.
ErsRep	Ers	Report an error.
DuiTokenInit	Dui	Initialise a DuiTokenContextType.
DuiTokenNext	Dui	Return the next token.
DuiEXecuteCmd	Dui	Execute a Dits command
ArgNew	Arg	Create an argument structure.
ArgPutString	Arg	Put a string into an argument.
MessGetMsg	Mess	Get the text associated with a message code.

External values used: none

Prior requirements: none

See Also: The Dits Specification Document, DuiExecuteCmd(3), DuiInteractive(3).

Support: Tony Farrell, `AAO`

E.3 DuiDetailsInit — Initialise a Dui Details structure.

Function: Initialise a Dui Details structure.

Description: To avoid users of the DuiExecuteCmd structure having to clear all the fields they are not using, this routine can be called to do so.

Language: C

Call:
(Void) = DuiDetailsInit (Details)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) Details (DuiDetailsType *) The action details to initialise. The default message type is set to `OBEY`, all other fields are cleared.

Include files: Dui.h

External functions used: none

External values used: none

Prior requirements: none

See Also: The Dits Specification Document, DuiExecuteCmd(3).

Support: Tony Farrell, `AAO`

E.4 DuiExecuteCmd — Execute a dits command

Function: Execute a dits command

Description: This routine dispatches a Dits Command to a specified task, under Uface context. It sets up its own handlers for the response to the message and outputs resulting messages using Ers and MsgOut.

The intention is to provide facilities normally required to user interface routines. This routine if fully reentrant allowing user interfaces to dispatch dits commands simply by calling this routine at will, regardless of if there are already commands outstanding.

The normal use of this routine would be to call it and then wait for messages using DitsMainLoop or DitsReceive

The address of a DuiDetailsType structure (details) is supplied by the user. This gives complete details about the action to be initiated.

Assuming DuiDetailsInit has been called for the details structure, the following are the only fields in details which must be setup.

Action	(Char [20])A string giving the name of the action to initiate
TaskName	(Char [80) A string giving the name of the task to which the action is to be sent, using the form task@node.
MessageBytes	(long int) As per DitsGetPath argument of the same name.
ReplyBytes	(long int)As per DitsGetPath argument of the same name.

The following fields are optional

MsgType	(DitsMsgType) Set to one of the message type codes accepted by DitsInitiateMessage, in its message argument, type item.
ArgId	(SdsIdType) Id of an argument structure to be sent with the message
ArgFlag	(DitsArgFlagType)What to do with the Argument - see the flag argument to DitsPutArgument. Default value `DITS_ARG_DELETE`
Node	(char [40]) The node on which the task resides. Assumes TaskName contains only the task name not the node.
SuccessHandler	(DuiHandlerType) A routine which is called if the action successfully completes. If it is not supplied or returns 0, then Dui will not output the results of the action.
ErrorHandler	(DuiHandlerType) A routine which is called if the action completes with an error. If it is not supplied or it returns 0, then Dui will output the results of the action.
TriggerHandler	(DuiHandlerType) A routine which is called a trigger message is received. If it is not supplied or it returns 0 then Dui will output an message about the trigger.
InfoHandler	(DuiHandlerType) A routine which is called a Information or Error message is received. If it is not supplied or it returns 0 then Dui will output the details.
CompleteHandler	(DuiCompleteHandlerType) A routine which is called whenever the action completes. It is called after the message has been handled either by Dui or a user supplied handler. The details structure is not required after this call and therefore can be deleted by this routine if it was dynamically allocated.
UserData	(void *) The user can store anything required here.
MaxMessages	(long int) As per DitsGetPath argument of the same name. Default 1.
MaxReplies	(long int) As per DitsGetPath argument of the same name. Default 1.
GetPathTimeout	(long int) The timeout to apply to Get Path operations. Default -1 = no timeout.
Timeout	(long int) The timeout to apply to the actual message. This is an integer number of seconds. Default -1 = no timeout.
Logging	(int) If true, enable logging of messages. This is an integer number of seconds. Will be enabled by default if `DRAMA`'s `DITS_LOG_LIBS` logging level is defined. Logging will be done to stderr or if a logging system is defined to the same place as DitsLogMsg(). See DitsLogMsg(3) for details.

The handler routines are only called if DuiExecuteCmd manages to initiate a transaction (either a GetPath transactions or, if the path if available immediately, the actual message transaction), otherwise this routine just returns with status set.

When a timeout occurs, it is treated as an error and the error handler invoked (if supplied), followed by the completion handler (if supplied). Note that the DitsGetReason() will return a reason of `DITS_REA_RESCHED` in these cases and a status of `OK`.

This following items in details are set by this routine and can used in the handler routines

MsgTypeString	(char *) A string which indicates the message string. This is used internally in messages. It will contain one of "Action", "Kick of Action", "Set of Parameter", "Get of ParameteR", "Control Message".
path	(DitsPathType) The path to the task. This may not be valid if the GetPath operation has failed or timed out.
ActionTimeoutFlag	(long int) Set true if the handler is being called as a result of the action timeout being triggered. It can be used to differentiate between action timeouts and get path timeouts (both of which will return a reason of `DITS_REA_RESCHED` from DitsGetReason().
SendCur	(int) Set true to the `DITS_M_SENDCUR` flag when initiating messages.
ReportLoss	(int) Set true to the `DITS_M_REP_MON_LOSS` flag when initiating messages.

Language: C

Call:

(Void) = DuiExecuteCmd (details, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) details (DuiDetailsType *) Details of command to execute. This structure must not be reused or released until the command completes (CompleteHandler called).

(!) status (StatusType *) Modified status.

Include files: Dui.h

External functions used:

DitsDeltatime	Dits	Convert a time to internal format
DitsErrorText	Dits	Get the text associated with a error message.
DitsGetArgument	Dits	Get the id of the argument structure assocaited with the current entry
DitsGetReason	Dits	Get the reason for an entry
DitsGetCode	Dits	Get the code associated with an entry
DitsGetPath	Dits	Get the path to a task
DitsInitiateMessage	Dits	Initiate a message to another task
DitsPutRequest	Dits	Put action return request
DitsUfaceCtxEnable	Dits	Enable Uface context
DitsUfaceRespond	Dits	Respond to messages for user interface
DitsUfaceTimer	Dits	Setup a timer
MsgOut	Dits	Output an informational message
ErsOut	Ers	Output a error message
strncpy	Crtl	Copy one string to another
strncat	Crtl	Concentrate one string onto another

External values used: none

Prior requirements: none

See Also: The Dits Specification Document, DuiDetailsInit(3).

Support: Tony Farrell, `AAO`

E.5 DuiInteractive — Setups up a Dits loop which also accepts interactive commands

Function: Setups up a Dits loop which also accepts interactive commands

Description: This routine uses the DitsAltInLoop routine to setup a program to handle both incomming Dits messages and input from a terminal device.

DuiInteractive hides the terminal i/o details and passes command lines to the callback routine. Note that no processing takes place on the command line, it is even handed empty lines. It is up to the callback procedure to process it correctly. See DuiCommand for a basic command callback.

When prompting is enabled, this routine will use DitsUfacePutErsPut and DitsUfacePutMsgOut to redirect the Ers and MsgOut messages though its own handlers. This enables correct handling of prompting after output.

Language: C

Call:

(Void) = DuiInteractive (CommandCallback, client_data, input, output, error, prompt, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

- **(>) CommandCallback (DuiCommandCallbackType)** The command callback routine
- **(>) client_data (void *)** Data for the command callback
- **(>) input (`FILE *`)** The input device. If null, use stdin.
- **(>) output (`FILE *`)** The output device. If null, use stdout
- **(>) error (`FILE *`)** The error output device. If null, use stderr.
- **(>) prompt (char *)** Input prompt string. If Null, don't prompt.
- **(!) status (StatusType *)** Modified status.

Include files: Dui.h

External functions used:

External values used: None

Prior requirements: The input device must be a tty, otherwise this routine will not work correctly.

See Also: The Dits Specification Document, DuiExecuteCmd(3), DuiCommand(3).

Support: Tony Farrell, `AAO`

E.6 DuiLoad — Load a dits program.

Function: Load a dits program.

Description: This routine initiates a load operation, under Uface context. It sets up its own handlers for the response to the load message and outputs messages using Ers and MsgOut when responses come back.

The intention is to provide facilities normally required by user interfaces. This routine is fully reentrant allowing use interfaces to load programs simply by calling this routine at will.

The normal use of this routine would be to call it and wait for messages using DitsMainLoop or DitsReceive.

The address of a DuiLoadDetailsType structure (details) is supplied by the user. This gives complete details about the program to be loaded.

Assuming DuiLoadDetailsInit has been called for the details structure, the following are the only fields in details which must be setup. See DitsLoad for more details about fields in this structure which correspond to DitsLoad arguments.

ProgramName	(Char [64]) The null terminated name of the program too load.

The following fields are optional

Node	(Char [40]) The machine on which to load the program. Default is the local machine.
ArgString	(Char [256]) The arguments to the loaded task.
Flags	(Long Int) The load flags. A mask of the following possiblilties - `DITS_M_ARG_SPLIT`, `DITS_M_REL_PRIO`, `DITS_M_ABS_PRIO`, `DITS_M_SYMBOL`, `DITS_M_NAMES`, `DITS_M_BYTES`.
ProcessName	(Char [15]) The name to be given to the process.
Decode	(Char [100]) The argument decode string.
Priority	(int) The relative or absolute proiority, if an appropriate flag is set.
Bytes	(int) Specifies the amount of memory to be used by a task. Only used if `DITS_M_BYTES` flag is set.
LoadedHandler	(DuiHandlerType) A routine which is called if the load completes and the loaded task registers with Imp. If it is not supplied or returns 0, then Dui will output details of the load using MsgOut.
LoadFailedHandler	(DuiHandlerType) A routine which is called if the load completes with an error. If it is not supplied or it returns 0, then Dui will output a message
LoadExitHandler	(DuiHandlerType) Called if the task was successfully loaded and has now exited, with or without an error.
CompleteHandler	(DuiCompleteType) A routine which is called when the load transaction completes. It is called after the message has been handled either by Dui or a user supplied handler. The details structure is not required after this call and therefore can be deleted by this routine if it was dynamically allocated.
UserData	(void `*`) The user can store anything required here.
Timeout	(long int) The timeout to apply to the load. Default -1 = no timeout.
Logging	(int) If true, enable logging of messages.

The handler routines are only called if DuiLoad manages to initiate a transaction. Otherwise this routine just returns with status set.

When a timeout occurs, it is treated as an error and the load failed handler invoked (if supplied), followed by the completion handler (if supplied). Note that the DitsGetReason() will return a reason of `DITS_REA_RESCHED` in these cases and a status of `OK`. Note, the timeout is cancelled when a successfull load occurs, not when the task exits.

Language: C

Call:
(Void) = DuiLoad (details, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`!`) details (DuiLoadDetailsType `*`) Details of program to load. This structure must not be reused or released until the command completes (CompleteHandler called).

(`!`) status (StatusType `*`) Modified status.

Include files: Dui.h

External functions used:

DitsDeltatime	Dits	Convert a time to internal format
DitsErrorText	Dits	Get the text associated with a error message.
DitsGetReason	Dits	Get the reason for an entry
DitsGetCode	Dits	Get the code associated with an entry
DitsLoad	Dits	Load a task.
DitsPrintReason	Dits	PRint the reason for an entry
DitsUfaceCtxEnable	Dits	Enable Uface context
DitsUfaceRespond	Dits	Respond to messages for user interface
DitsUfaceTimer	Dits	Setup a timer
MsgOut	Dits	Output an informational message
ErsOut	Ers	Output a error message
strncpy	Crtl	Copy one string to another

External values used: none

Prior requirements: none

See Also: The Dits Specification Document, DuiLoadDetailsInit(3).

Support: Tony Farrell, `AAO`

E.7 DuiLoadDetailsInit — Initialise a Dui Load Details structure.

Function: Initialise a Dui Load Details structure.

Description: To avoid users of the DuiLoad structure having to clear all the fields they are not using, this routine can be called to do so.

Language: C

Call:
(Void) = DuiLoadDetailsInit (Details)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) Details (DuiLoadDetailsType *) The action details to initialise.

Include files: Dui.h

External functions used: none

External values used: none

Prior requirements: none

See Also: The Dits Specification Document, DuiLoad(3).

Support: Tony Farrell, `AAO`

E.8 DuiLogEntry — Routine called to load the details of an entry to an action

Function: Routine called to load the details of an entry to an action

Description: Call this routine to log details of an entry to an action using either stderr or the facilities provides by the task logging system, if any. The specified prefix is added to each line.

Language: C

Call:
(void) = DuiLogEntry (prefix,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) prefix (const char *) Prefix for each line to be output.

(!) status (StatusType *) Modified status

Include files: dui.h

Support: Tony Farrell, `AAO`

E.9 DuiTokenInit — Initialise a Dui String Token Context.

Function: Initialise a Dui String Token Context.

Description: The DuiToken routines provide a similar facility to the C run time library routine strtok. Unlike strtok, these routine provide a facility which is re-entrant and does not change the subject string.

DuiTokenInit will intialise a DuiTokenContextType variable to the specified string. You can then uses DuiTokenNext to return the tokens.

Language: C

Call:

(Void) = DuiTokenInit (string, context)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) string (char `*`) The string to be divided into tokens.

(`!`) context (DuiTokenContextType `*`) The context to be initialised.

Include files: Dui.h

External functions used:

malloc	`CRTL`	Allocate memory
strlen	`CRTL`	Obtain the length of a string
strcpy	`CRTL`	Copy one string to another.

External values used: none

Prior requirements:

See Also: The Dits Specification Document, DuiTokenNext(3), DuiTokenShut(3).

Support: Tony Farrell, `AAO`

E.10 DuiTokenNext — Initialise a Dui String Token Context.

Function: Initialise a Dui String Token Context.

Description: The DuiToken routines provide a similar facility to the C run time library routine strtok. Unlike strtok, these routine provide a facility which is re-entrant and does not change the subject string.

DuiTokenNext takes the address of a context variable initialised by DuiTokenInit. It returns a pointer to the next token in the string. You generally call this routine repeatly to obtain successive tokens.

Language: C

Call:
(char *) = DuiTokenNext (context, delim)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(!) context (DuiTokenContextType *) The context initialised by DuiTokenInit

(>) delim (char) The token delimiter. This points to the delimiter character. The delimiter character may differ from call to call. (Note, strtok has a string here, not a character)

Function Value: The address of the next token or zero if no token was found.

Include files: Dui.h

External functions used: none

External values used: none

Prior requirements:

See Also: The Dits Specification Document, DuiTokenInit(3), DuiTokenShut(3).

Support: Tony Farrell, AAO

E.11 DuiTokenShut — Tidy up a Dui String Token Conext.

Function: Tidy up a Dui String Token Conext.

Description: The DuiToken routines provide a similar facility to the C run time library routine strtok. Unlike strtok, these routine provide a facility which is re-entrant and does not change the subject string.

The DuiTokenShut routine releases memory allocated by DuiTokenInit.

Language: C

Call:
(Void) = DuiTokenShut (context)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(!) context (DuiTokenContextType *) The context to be released.

Include files: Dui.h

External functions used:

free	Crtl	Free allocated memory

External values used: none

Prior requirements:

See Also: The Dits Specification Document, DuiTokenInit(3), DuiTokenNext(3).

Support: Tony Farrell, `AAO`

E.12 DuiVxWorksInit — Initialise a VxWorks main routine

Function: Initialise a VxWorks main routine

Description: This function takes a VxWorks function name and argument string and returns appropriate argc and argv variables.

The argv array is dynamally allocated and should be freed when you are finished with it. The supplied arguments strings (args) is modified, argv will point to parts of this string. The exception to this is argv[0], which points to dynamically allocated memory. So you should explictly free argv[0] and argv.

Language: C

Call:

(Void) = DuiVxWorksInit (function,args, argc, argv, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) function (`UINT`) Pointer to main function. This is used to obtain the value for argv[0].

(`!`) args (char `*`) Argument string. Modified

(`<`) argc (int `*`) argc returned here

(`<`) argv (char `***`) Argv returned here.

(`!`) status (StatusType `*`) Modified status

Include files: Dui.h

External functions used:

DuiTokenInit	Dui	Initialise a string token
DuiTokenNext	Dui	Return next token
DuiTokenShut	Dui	Release a string token
malloc	Crtl	Allocate memory
symFindByValue	VxWorks	Find a symbols name using its value.

External values used: sysSymTbl (VxWorks) The VxWorks system symbol table.

Prior requirements: none

See Also: The Dits Specification Document, DuiTokenInit(3), DuiVxWorksInit(3).

Support: Tony Farrell, `AAO`

F Dmon routines

These are out of date routinesused for communicating with the `UDISPLAY` ADAM U-task. They will are some stage be remove and should not be used in new code. See [8] for details.

F.1 DmonCommand — Handle the `UMONITOR` command, Obsolete.

Function: Handle the `UMONITOR` command, Obsolete.

Description: User Dits tasks should call this routine when the `UMONITOR` action is received.

The path to the task which initiated the action is remembered to enable messages to be sent to it using the DmonParameter and DmonState calls.

Language: C

Call:
(Void) = UmonCommand (status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`!`) status (StatusType `*`) Modified status.

Include files: Dmon.h

External functions used:

DitsGetSeq	Dits	Get the action sequence value.
DitsPutRequest	Dits	Request for when action exits.

External values used: DitsTask - Details of the task.

Prior requirements: None

Support: Tony Farrell, `AAO`

F.2 DmonParameter — Notify the a monitor task of a change in a parameter value, Obsolete.

Function: Notify the a monitor task of a change in a parameter value, Obsolete.

Description: Send a message to the initiator of the `UMONITOR` action giving the name and new value of the specified parameter.

The parameter name must correspond to a parameter in the interface file being used by the `UDISPLAY` task and the value string but be able to be converted to the type of the corresponding parameter.

If DmonCommand has not been called or the task is running as a Utask task then no message is sent.

Language: C

Call:

(Void) = DmonParameter (parameter, value , status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) parameter (Char `*`) The parameter name

(`>`) value (Char `*`) The parameter value.

(`!`) status (StatusType `*`) Modified status.

Include files: Dmon.h

External functions used:

strcpy	`CRTL`	Copy one string to another.
strcat	`CRTL`	Concentrate one string onto another.
ArgNew	Arg	Create a new argument structure.
ArgPutString	Arg	Put a string item.
Dits___SendTap	`DITS` internal	Send a message to the originator of an action.
SdsDelete	Sds	Delete an Sds item.
SdsFreeId	Sds	Free An Sds identifier.

External values used: None

Prior requirements: None, but unless DmonCommand has been called, nothing happens

Support: Tony Farrell, `AAO`

F.3 DmonState — Change the state of the Utask display task, Obsolete.

Function: Change the state of the Utask display task, Obsolete.

Description: Send a message to the initiator of the `UMONITOR` action specifing a new state.

If DmonCommand has not been called then no message is sent.

Language: C

Call:

(Void) = DmonState (state, status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(`>`) state (Char `*`) The new state

(`!`) status (StatusType `*`) Modified status.

Include files: Dmon.h

External functions used:

strcpy	`CRTL`	Copy one string to another.
strcat	`CRTL`	Concentrate one string onto another.
ArgNew	Arg	Create a new argument structure.
ArgPutString	Arg	Put a string item.
Dits___SendTap	`DITS` internal	Send a message to the originator of an action.
SdsDelete	Sds	Delete an Sds item.
SdsFreeId	Sds	Free An Sds identifier.

External values used: None

Prior requirements: None, but unless DmonCommand has been called, nothing happens

Support: Tony Farrell, `AAO`

G Programs

This section documents the various programs which make up the Dits package.

G.1 dits_netclose — Shutdown dits network tasks

Function: Shutdown dits network tasks

Description: Shutdown IMP_Master and `ADITS` (if running).

Author: Tony Farrell, `AAO`

G.2 dits_netstart — Startup Dits network communications processes.

Function: Startup Dits network communications processes.

Description: Starts the processes required for network operations with Dits. These are the Imp Master task and if Starlink `ADAM` is supported, the `ADITS` Adam to Dits converter task. The Imp Master task will automatically load the transmitter and receiver tasks.

Author: Tony Farrell, `AAO`

G.3 ditscmd — Sends a command to a Dits task and waits for the response.

Function: Sends a command to a Dits task and waits for the response.

Description: This program provides a simple user interface to a Dits task. It first gets the path to the specified Dits task and sends an obey message with the specified name and arguments. It waits for the response, outputting the details of any Error for Informational messages received from the task. When the subsidiary action completes, `DITSCMD` exits, returning the completion status of the subsidiary action.

Synopsis: ditscmd [options] task action [action_arguments ...]

Under VxWorks, all options and arguments should be enclosed in the one string.

Command arguments:

task The name of the task to send the message to. If a remote task use the format task@node, where node is the internet node name of the machine the task is running on.

action The name of the action or parameter to in the task

action_arguments All other arguments are considered arguments to the task, and are encoded as character strings in an argument structure. If the string contains an equals sign, it is assumed that the characters before the equals sign are the parameter name, and the characters after are a string value. Otherwise each argument has the name

`Argument<n>`

Where `<n>` is the number of the argument, starting at 1. The -z option can be used to change the equals sign to something else.

If doing a parameter get operation, the arguments are names of extra parameters to fetch.

Options:

-b size Depreciated. Size is the total message buffer size. Default 15000. This size should be about bigger then that required by the reply values specified by the "-m" option. This is now sized automatically if you change the reply buffer sizes - the relevant part of the "-m" spec. In that case the global buffer will always be set to be at least large enough to set up the reply buffer.

-m n1:n2:n3 Sets message buffer sizes for the connection. n1 = Messages bytes allowed for message to be sent n2 = Messages bytes allowed for return messages n3 = Number of return messages which may occur Default is 800:800:10 which should be sufficient for most command line level tasks. The first item is actually overridden to a size appropriate for the argument to be sent - its use here is historical.

Depreciated - in most cases -r will do what you need

-r n Sets the reply buffer size. Sets max replies to 1 This replaces -m for most use cases.

-n name The name this task should register itself as. Default is `DITSCMD_<n>` where `<N>` is a hex representation of the process id. This option may be required if you intend to have several versions of this program outstanding at one time.

-o Send obey message (default).

-k Send kick message.

-s Send set message.

-g Send get message. (Actually multiple parameter get message, but this works ok one only one parameter is specified).

-c Send control message.

-p Send parameter monitor message.

-l Send multiple parameter get message (redundant, you can use -g).

-v Report responses verbosely using SdsList, not briefly using ArgToString. Overrides any -w option.

-w Write any returned Sds structure to the file ditscmd_out.sds. Overrides any -v option. On successfully completion program will exit quietly after writing the file. Can be replaced by -t when not on VxWorks

-t <file> Write any returned Sds structure to the file <file>. Overrides any -v option. On successfully completion program will exit quietly after writing the file. Not available on VxWorks

-a <file> Read the obey arguments from the specified file rather then the command line. Non-VxWorks versions only. Any extra arguments will cause an error.

-z <c> Change from using "equals" in the Name=value argument naming convention to using the specified character. Only the first character of the value is used.

-q Special fetch of parameters. Instead of outputting the result using MsgOut, it is directly output to stdout. This makes it easier to fetch parameter values in scripts. Is overridden by -w, -v and any flag which causes another message type to be sent.

-x If the sub-task exits, then don't return an error code from this program.

– Stop processing options. You will need to use this if your arguments contain negative numbers or other values prefixed by a dash.

-e n Specifies the expected return code for the message. This allows you to specify error code values which are the expected status from the message and if that is returned then ditscmd returns ok, otherwise it returns an error even if status is ok. Often used to check to expected error conditions in regression testing. 'n' is a positive integer. Not supported in VxWorks

-h Output help on this program.

Language: C

Support: Tony Farrell, AAO

G.4 ditsgetinfo — Return details about DRAMA tasks.

Function: Return details about DRAMA tasks.

Description: This program is used to return various details about running DRAMA tasks.

Synopsis: ditsgetinfo [options] nametype

Command arguments:

nametype Either the task name of task type, depending on the options -bytype and -byname

Command options:

-n The name for this task

-b The global buffer size

-bytype The nametype argument specifies the type of the task to look for. Default is -byname

-byname The `nametype` argument specifies the name of the task to look for. This is the default

-remote Task may be on the specified remote node. We try to locate it first. Won't work with -bytype. Specify node name using normal task@node format. This requires that the `DRAMA` networking be running on the current and target machines. Note, you `CANNOT` locate remote with the same names as tasks already running on the current machine. This includes the `IMP` networking tasks.

The following options are mutually exclusive and the default is -running.

-running Just return a success status if the task is running. If it is not, return error code 1 (but not other message unless some other error occured).

-type If task is running, output its type.

-descr If task is running, output its description.

-name If task is running, output its name.

-full Output full details.

Language: C

Support: Tony Farrell, `AAO`

G.5 ditsgetinfo — Return details about `DRAMA` tasks.

Function: Return details about `DRAMA` tasks.

Description: This program is used to return various details about running `DRAMA` tasks.

Synopsis: ditsgetinfo [options] `nametype`

Command arguments:

`nametype` Either the task name of task type, depending on the options -bytype and -byname

Command options:

-n The name for this task

-b The global buffer size

-bytype The `nametype` argument specifies the type of the task to look for. Default is -byname

-byname The `nametype` argument specifies the name of the task to look for. This is the default

-remote Task may be on the specified remote node. We try to locate it first. Won't work with -bytype. Specify node name using normal task@node format. This requires that the DRAMA networking be running on the current and target machines. Note, you CANNOT locate remote with the same names as tasks already running on the current machine. This includes the IMP networking tasks.

The following options are mutually exclusive and the default is -running.

-running Just return a success status if the task is running. If it is not, return error code 1 (but not other message unless some other error occured).

-type If task is running, output its type.

-descr If task is running, output its description.

-name If task is running, output its name.

-full Output full details.

Language: C

Support: Tony Farrell, AAO

G.6 ditsloadcmd — Loads a program using the DRAMA task loading facilities.

Function: Loads a program using the DRAMA task loading facilities.

Description: Attempts to load a program using the DRAMA task loading facilities. This provides a couple of potential advantages over just running a program. You can load a program remotely and you can wait for a notification that it has registed correctly with DRAMA.

Synopsis: ditsloadcmd [options] program

Command options:

-n The DRAMA task name for ditsloadcmd.

-b The DRAMA global buffer size for ditsloadcmd. You are unlikely to need to set this.

-waitexit Wait until the loaded program exits. By default we only wait until it loads

-nowait Don't wait for task loading.

-node name The node on which the program should be loaded. If not supplied, load on the machine ditsloadcmd is running on.

-argument string Command line arguments to the loaded program

-process name The process name to be given the the program Only used if the machine on which the target task is running supports process names. E.g. VMS and VxWorks.

-bytes integer The number of bytes to allocated for the program.

-priority integer The requested priority for the program

-timeout integer Load timeout value

-split Split arguments using decode string (not yet supported)

-decode string Argument decode string (not yet supported).

-relprio Priority is relative to parent

-absprio Priority is absolute

-symbol For `VMS` target nodes, program is a symbol

-prog For `VMS` target nodes, program is a program name

-names For `VMS` target, inherit logical names and symbols. For Unix target, inherit environment.

-mayload If the master task is not loaded and program is to be loaded locally, ditsloadcmd may load it itself.

-adam Only supported if compiled with `ADAM` support. Says we are loading an Adam task and if waiting for the load to complete, wait for an `ADAM` load not a `DRAMA` load. The two styles are mutually exclusive. If not timeout was supplied, then 20 seconds is used. Note, this does not work for remote loads.

-h Output this help.

Language: C

Support: Tony Farrell, `AAO`

G.7 ditssavearg — Save a action `DRAMA` argument to a file for use by "ditscmd -a"

Function: Save a action `DRAMA` argument to a file for use by "ditscmd -a"

Description: This program is used to save the value of an action argument to a file such that it can be sent by "ditscmd -a"

Synopsis: ditssavearg [`<task_name>`] [`<action_name>`] [`<file_name>`]

Command arguments: `<task_name>` The name this task should register under. Defaults to `DITSSAVEARG`

`<action_name>` The action name to implement. Defaults to `SAVEARG`. `<file_name>` The name of the file to write the argumen too. Defaults to ditsarg.sds.

Command options:

-b `<bytes>` The global buffer size. Defaults to 7000.

-k Action should kick handler will implement the save. The obey handler will just reschedule and the user can then send a kick.

Language: C

Support: Tony Farrell, `AAO`

G.8 dumpana — Analyzes a DRAMA task message log.

Function: Analyzes a DRAMA task message log.

Synopsis: dumpana file [first [last]]

Description: A DRAMA task can generate a log of all messages sent and received using DitsDumpImp(). This command analyzes such a dump.

first is the number of the first message to be dumped. last is the number of the last message to be dumped. By default, all messages are dumped.

In the current version, only if the first message is number 1 are the notes about invalid transaction id's valid.

G.9 tasks — Lists known DRAMA tasks.

Function: Lists known DRAMA tasks.

Description: This program lists the known DRAMA tasks.

Synopsis: tasks

Language: C

Support: Tony Farrell, AAO

References

[1] Tony Farrell, AAO. *05-Aug-1993, Guide to Writing Drama Tasks.* Anglo-Australian Observatory **DRAMA** Software Document number 3.

[2] Keith Shortridge, AAO. *22-Nov-1994, Interprocess Message Passing System.* Anglo-Australian Observatory **DRAMA** Software Document number 8.

[3] Jeremy Bailey , AAO. *31-Oct-1994, Self-defining Data System.* Anglo-Australian Observatory.

[4] Tony Farrell, AAO. *16-Mar-1993,* **DRAMA** *Error reporting System.* Anglo-Australian Observatory **DRAMA** Software Document number 4.

[5] Tony Farrell, AAO. *23-Feb-1995, A portable Message Code System.* Anglo-Australian Observatory **DRAMA** Software Document number 6.

[6] Tony Farrell, AAO. *29-Jul-1993,* **DRAMA** *Software Organisation.* Anglo-Australian Observatory **DRAMA** Software Document number 2.

[7] Tony Farrell, AAO. *29-Jul-1993, Create* **DRAMA** *Makefiles.* Anglo-Australian Observatory **DRAMA** Software Document number 11.

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