ANGLO-AUSTRALIAN OBSERVATORY AAO/GIT_SPEC_9
DRAMA Software Report 9
Version 0.4

Tony Farrell
18-Jan-94

Generic Instrumentation Task Specification

Contents

Revisions:

V0.4 18-Jan-1993 Introduce **GitMonitor** routines, **GitPutDelay()** and **GitPutDelayPar()**. Add **GitArgGetStruct()**. Various documentation errors are corrected.

1 Introduction

This document defines some standards that should be adhered to by all **DRAMA** based instrumentation tasks [1].

The main reasons for defining such standards are

- to try to minimise unnecessary differences between instrumentation tasks. For example, it is simply unnecessary that task A should use action name **SHUTDOWN** to exit whilst task B uses **EXIT**.
- to provide facilities in all tasks that can be assumed to exist in all cases. For example, all tasks should perform some form of simulation mode that permits them to be run in some form in the absence of their hardware.
- to define those interfaces that are not already well-defined by the **DRAMA** environment system.

This document assumes you have read the "Guide to Writing **DRAMA** Tasks" [1].

Note that we are not imposing standards on all **DRAMA** tasks, just on AAO tasks which control instrumentation.

In addition, various routines which either help implement this standard or provide general help to **DRAMA** task authors are introduced.

1.1 Instrumentation Tasks

An instrumentation task is a task responsible for controlling a major sub-system of an instrumentation system, such as a detector or spectrograph or a combination of these systems. Almost all **DRAMA** tasks would fall under this definition. An exception is where small stripped down tasks are used by an instrumentation task to perform special functions, such as to hide blocking I/O operations (since the actual instrumentation task should always be able to accept messages). These small tasks need not obey this standards. All other instrumentation tasks should!

2 General

2.1 Task Initialisation

Try to do as little as possible in the **main()** routine of your program. If anything goes wrong in this routine, it is not always possible to direct messages to the correct user interface. In most cases **main()** should only need to -

1. Initialise the parameter system using **SdpInit()**.

[1]This document is the **DRAMA** equivalent of the ASD document - "General D-task Specifications". That document is specific to ADAM based tasks.

2. Initialise **Dits** using **DitsInit()**.
3. Tell **Dits** about the parameter system using **DitsPutParSys()**.
4. Put **Dits** action handlers using **DitsPutActionHandlers()**.
5. Create parameters using **SdpCreate()**.
6. Enable the message facility of the task, using **MessPutFacility()**.
7. Call **DitsMain()** and **DitsStop()**.

Four and five are often done by package activation routines. See [1], [3] and [4] for more details.

The real job of initialising the Instrument should be performed in a separate **INITIALISE** action, which is normally the first action obeyed by the task.

Task names should be upper case only and less then 15 characters long. This will allow the task to be controlled from an ADAM based user interface.

2.2 Action names

Action names should be upper case and no more then 15 characters. **Dits** allows longer names and will distinguish between upper and lower case characters. You should avoid these features since, if you use them, it may not be possible to invoke such actions from ADAM based user interfaces and control tasks.

2.3 Action Arguments

The **DitsGetArgument()** routine allows an action to fetch the **Sds** id of its argument. When an action is exiting, any argument value put with **DitsPutArgument()** will be made available to the invoking task. Such arguments will normally have been constructed using the **Arg** routines supplied as part of **Sds**. The **Arg** and **Sds** routines can be used to access arguments.

Many actions expect arguments to be supplied. In order to use the **Arg** routines to access them, you must know what they are called. The various user interfaces will construct arguments using the name `Argument`*n* where *n* is the argument number starting at one.

Most arguments are optional. If not supplied the action should use an appropriate default. Mandatory arguments are noted as such and actions with mandatory arguments should return an error if the argument is not supplied.

See [1] and [5] for more details.

2.4 Parameters

Instrumentation tasks should enable and use the **Sdp** parameter system supplied as part of **Dits**. In general, parameter names should be upper case and no more then 15 characters to allow them to be seen by ADAM tasks. This is not required for Structured parameter items which cannot in any case be handled by ADAM tasks.

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

2.5 Completion of Actions

An action never completes until whatever it initiates has completed. This means that an **EXPOSE** action completes only when the exposure that it initiates completes, a **READOUT** action completes only when the readout that it initiates is complete, and a **POLL** action completes only when the polling that it enables is disabled.

2.6 Error Reporting and Status Returns

Tasks will define their own facility codes and will use the **MESSGEN** utility to define status values (see [4] for details) and associated text (a range of facility numbers has been allocated to AAO and you can allocate new ones by modifying the first message in the PROG bulletin board folder). Don't expect control tasks to test for specific status responses. Instead, use the severity bits. A control task will assume that an action has succeeded if it returns a success, informational or warning status. If it returns an error status then the action has failed but it may be possible to retry it.

The argument returned by an action may contain information which helps the control task to decide how to recover from the error. The specification for the action in question should state what use is made of the value string (for example, an action might return an error status together with an integer encoded into the value string — a value of 0 might mean "retry the last action" and a value of 1 might mean "retry the action before last and then retry the last one"). A fatal error status always means that the control task should abort whatever sequence of actions it is currently executing.

When writing instrumentation tasks, be aware that the user interface will always have output any text sent using **MsgOut()** or the **Ers** routines (see [2]) and will always translate the returned status. Try not to duplicate too much information. Specifically, please adhere to the following rules:

1. Terminate messages with full stops.
2. Under **DRAMA**, the control task can find out the originating task of a message output using **MsgOut()** or the one **Ers** routines. This will normally be prefixed to any message when it is output. Message texts should take account of this by not including the task name in the message.
3. Use the **DitsErrorText()** routine to translate status values.

3 Standard Actions

All Instrumentation tasks should provide the following actions -

INITIALISE Initialise the task. This involves setting up such things as simulation level and hardware. It must be the first action which is executed and except where noted, other actions should refuse to do anything if the task has not been initialised.

CTRLC Interrupt the task. The interpretation of such an interrupt is entirely up to the task and will usually be state-dependent.

DUMP_LOG Dump the internally-saved recent history to the log file.

The optional argument should be a character string which will be written to the log file as part of the dump. It specifies the reason for the dump and is normally directed to readers of the log.

EXIT Exit tidily from the task. The task can choose to reject this command if it is not in an appropriate state. Equally, it can reschedule until it is in the correct state. That is up to the task.

An argument is optional. It's type and interpretation is task specific.

LOG_LEVEL Set logging level. Use to turn logging levels on or off. When a task is loaded, the initial logging level is determined as described in section 7.

The mandatory argument is a character string is is passed directly to the logging routines. See section 7 for more details.

POLL Begin polling instrument status. This action exists so that the task always has an active action to which asynchronously occurring errors can be reported. If **POLL** needs to report a problem to its originator, it can terminate returning a status in its completion message and the originator will normally re-issue the **POLL** command. It may also use **DitsTrigger()**.

POLL can receive a **KICK** message, which should cause **POLL** it to exit.

The calling task should not make assumptions about the internal actions of the task — **POLL** may not actually imply a polling loop but if there is no polling loop the action should not complete, since this can lead to its originator continually detecting that it has completed and continually re-issuing it. (**POLL** should put a request of **DITS_REQ_ASTINT** in this case, see **DitsPutRequest()** for details).

Takes one optional argument which should be the polling interval in milliseconds.

POLL_PARAMETER Change the value of the poll parameter (with immediate effect). The mandatory argument is the new polling interval in milliseconds.

RESET A complete reset of the system. Will stop anything currently in progress, but not the **POLL** action, which should conceptually stop and restart itself. Most things will need re-loading. Note that this may be a time-consuming operation.

The optional argument is a character string indicating what sort of reset should be preformed. All tasks must support the following values:

SOFT Perform a partial reset. Exactly what is meant by this is a function of the instrument in question.

FULL Attempt to restore the instrument to the same state that it was in after the **INITIALISE** action completed.

HARD Reset hardware and then perform a FULL reset.

Individual tasks may support additional values in addition to these ones. If the argument is not supplied, the task should do a SOFT reset.

STATUS Get the current status of the task. This action returns an argument which described the status. See section 8 for more details.

SIMULATE_LEVEL Set the current simulation level. A **RESET** may be required for a change in level to take effect. When a task is loaded, the initial simulation level is determined as described in section 5 but this command may be sent prior to **INITIALISE** to change initial setting.

There are two arguments. The first is a mandatory character string specifying the simulation level. It should have one of the values specified in section 5. The second argument is optional. It specifies the time base of the simulation mode. A time base of 10 indicates simulation should take place ten times faster then normal. Likewise, a time base of 0.1 means they happen ten times slower.

UPDATE_NBD This action is required for compatability with ADAM control tasks (such as the Observer Control Task). The details are to be specified later.

4 Standard Parameters

There are some parameters that should be provided by all tasks. They exist to permit control tasks to find out about the task they are controlling, to enable configuration changes and to help in debugging.

LOG_LEVEL This parameter contains the current logging level. It's initial value is the default logging level. For more details see section 7.

SIMULATE_LEVEL This indicates the current simulation level. It's initial value is the default simulation level. For more details, see section 5.

TIME_BASE This indicates the current simulation time base. It's initial value is the default simulation time base. For more details, see section 5.

ENQ_DEV_TYPE The type of device that is controlled by the task. It is a character parameter and must have a default value. The value should be one of a set of defined upper-case keywords that indicate what sort of commands this task can be expected to respond to. Possible values are

GCT Generic Control Task

DCDT Detector Control D-Task

DHDT Data Handling D-Task

DRT Data Recording Task

TEL Telescope D-task

IDT Instrument D-Task

ENQ_DEV_DESCR A printable description of what this D-task does. It is a character parameter and must have a default value. It should be possible to concatenate this with **ENQ_VER_NUM** and **ENQ_VER_DATE** to give a pleasing summary message.

ENQ_VER_NUM The task version number. It is a character parameter and should have a default value . It should be a string of the form "Vn.m.p.q" and should be identical to the CMS class name corresponding to this version.

ENQ_VER_DATE The date that this version was created. It is a character parameter and should have a default value. It should normally be the creation date of the CMS class mentioned above in the form "dd-Mmm-yy".

ENQ_DEV_NUMITEM Required for compatability with ADAM based control tasks. Should be an integer with a value of zero.

5 Simulation

All tasks should provide some level of simulation. The minimum is that it should be possible to load the task in the absence of the hardware. The maximum is that, once loaded, the task should accept all commands and appear to be doing sensible things with them. All tasks should accept the **SIMULATE_LEVEL** action.

If a task does not support the requested simulation level or time base, it does not return an error message or status. Instead it does the best that it can and sets the actual values into the **SIMULATE_LEVEL** and **TIME_BASE** parameters, which can then be read by the sender of the command.

5.1 Use of Logical Names/Environment Variables

If, when initialising, the **SIMULATE_LEVEL** parameter has no value, the *subsys*_SIMULATE logical name (VMS) or environment variable (UNIX/VxWorks) will be translated to provide a value for it and this value will be set as the value of the parameter.

If, even after application of the above rules, no values for **SIMULATE_LEVEL** are found, **SIMULATE_LEVEL** defaults to NONE (which is defined below) and **TIME_BASE** defaults to 1.

5.2 Banning of Automatic Fall-back into Simulation Mode

It will not be acceptable to automatically enter some level of simulation as a result of a failure to talk to the hardware. This practice has been followed in the past by some tasks, but is dangerous in that it can lead to inadvertent use of simulation modes. It is now banned.

5.3 Standard Simulation Levels

The following standard values for **SIMULATE_LEVEL** are defined. The values are upper-case character strings. If individual authors require new ones they should get them added to this standard list.

NONE No simulation is being performed.

BASIC task will load but commands that interact with the instrument may fail. This is the minimum level that must be supported by all tasks.

COMMANDS Commands that interact with the instrument will work, but timing will not be realistic and responses from the instrument will not be simulated.

STATUS Commands will work and status responses from the instrument will be simulated.

FULL Full simulation is being performed. Commands and status responses will work and will take reasonably realistic amounts of time.

6 Task Responsiveness

A task should always be responsive to incoming messages. It must be possible to abort time consuming operations, which can only be done if the task remains responsive to incoming messages.

To remain responsive while performing a time consuming operation, an action has one of three options-

- An action can reschedule frequently. This is not too inefficient, it could be done say once a second in an appropriate part of a calculation.
- An action can check if messages are available. This is done using **DitsMsgAvail()**. If this returns a non-zero count then the action should reschedule. This technique is a bit more efficient, but possibly more complex to implement.
- One of the previous methods are preferred, but not always possible. If your tasks calls a time consuming routine over which it has no control (say a NAG routine), neither of the above techniques are possible. In this case, you should create a new process or process thread to handle the time consuming operation. You must then kill that process to abort the operation.

All Instrumentation tasks must respond to messages within a second.

7 Logging

To be specified.

8 Task Status

To be specified.

9 The Git package

A package is provided to help implement tasks which obey this standard.

The **Git** package also provides several other general purpose routines implementing functions commonly required by tasks. It is intended that any function which is found to be required by multiple instrumentation tasks but which does not fit in any other package should be included in **Git**.

It is recommend (but not required) that you make use of this package. The routines are documented in the appendix of this document.

9.1 GitActivate

The **GitActivate()** routine is is the core of the package. It is written using the object oriented techniques described in [1] and is used as part of an example in that document. The parameters specified above have appropriate defaults while the actions do the minimum possible to obey the standard.

In addition to the above actions, **Git** supports the **UMONITOR** action using the **Umon** routines specified in [3]. This allows the task to be run from the **UDISPLAY** task described in [6]. Also supported under VMS systems is the **DEBUG** action. When invoked, this action will activate the VMS debugger.

It is not necessary to use **GitActivate()** to make use of the rest of the routines in this package, although some required the **Sdp** parameter system to be enabled.

9.2 GitArg routines

The various routines starting with **GitArg** provide a wrap around for common command line argument handling. The routines available are

GitArgGetD() Gets a double floating point argument value with checking against a range.

GitArgGetI() Gets an integer argument value with checking against a range.

GitArgGetL() Gets a logical argument value with user supplied strings for true and false strings. For example, you can specify that “OPEN’ and “CLOSE” are to be considered true and false.

GitArgGetS() Gets a string argument with checking against acceptable values.

GitArgGetStruct() Returns the id of an Sds structure. The structure may have been provided directly as the argument to the action or by supplying the name of a file containing the structure.

These routines have a range of options, allowing for most required situations in getting arguments. The user supplies the SDS id of an argument structure (usually obtained using **DitsGetArgument()**) The routines first try to find the argument by name and if one of that name does not exist, it tries for an argument in the specified position. This allows for both general user interfaces (which don't know about argument names) and specific ones which will know the appropriate argument name.

The routine **GitArgNamePos()** is used by the above routine to get the id of an argument by name or position.

9.3 Parameters/Environment routines

The **GitGetEnvS()** routine returns the value of a logical name (VMS) or environment variable (UNIX/VxWorks). The routine **GitParEnvGetS()** first tries to find a value in a parameter. If it fails or returns a bad values, then it translates the specified logical name (VMS) or environment variable (UNIX/VxWorks). If this fails, the specified default is returned.

GitSimulation() sets the task simulation level in the required way.

9.4 GitPathGet routines

GitPathGetInit() and **GitPathGetComp()** are used as a pair. They simplify the getting of a path.

This is how you would get a couple of paths using **DitsGetPath()**. This example is taken from the control task example in [1].

It is a bit complex, but, it is the most efficient and flexible way of doing things.

```
static void CTestFindPaths(StatusType *status)
{
    if (*status != STATUS__OK) return;
/*
 *  Convert the timeout into a value appropriate for Dits.
 */
    if (TIMEOUT > 0)
        DitsDeltaTime(TIMEOUT,0,&timeout);

/*
 *  Clear the TaskDied status
 */
    TaskDiedStatus = STATUS__OK;
/*
```

```
 *  Initiate getting the paths to the TEA and COFFEE tasks.
 */
    DitsGetPath("TEA",MESSAGEBYTES,MAXMESSAGES,REPLYBYTES,MAXREPLIES,
                &teaPath,&teaTransid,status);
    DitsGetPath("COFFEE",MESSAGEBYTES,MAXMESSAGES,REPLYBYTES,MAXREPLIES,
                &coffeePath,&coffeeTransid,status);

    if (*status == STATUS__OK)
    {
/*
 *      We already have both paths, just go to the next stage
 */
        if ((teaTransid == 0)&&(coffeeTransid == 0))
        {
            DitsPutHandler(CTestStartActions,status);
            DitsPutRequest(DITS_REQ_STAGE,status);
        }
/*
 *      Otherwise, wait for the message with timeout if necessary.  Use
 *      CTestPathFound to respond.
 */
        else
        {
            if (TIMEOUT > 0)
                DitsPutDelay(&timeout,status);
            DitsPutHandler(CTestPathFound,status);
            DitsPutRequest(DITS_REQ_MESSAGE,status);
        }
    }
    else
        ErsRep(0,status,"Error trying to get paths to TEA and COFFEE");
}

/*
 * We get here when a path is found or if a timeout getting the paths occurs
 */
static void CTestPathFound(StatusType *status)
{
    DitsReasonType reason;
    StatusType reasonstat;
    DitsGetReason(&reason,&reasonstat,status);
/*
 *  If the reason is a reschedule message, then we have timed out. Exit the
 *  action, but not the task.
 */
    if (reason == DITS_REA_RESCHED)
    {
        *status = DITS__APP_TIMEOUT;
        ErsRep(0,status,"Timeout trying to get paths");
    }
/*
 *  If its a path not found message, then report a message and exit the
 *  action.
 */
    else if (reason != DITS_REA_PATHFOUND)
    {
        char name[DITS_C_NAMELEN];
        DitsTransIdType transid;
        DitsPathType path;
        DitsGetEntInfo(sizeof(name),name,&path,&transid,&reason,
                            &reasonstat,status);
        *status = reasonstat;
        ErsRep(0,status,"Failed to get path to task %s: %s",name,
                                         DitsErrorText(*status));
    }
/*
 *  Successful find of a path.  Check which one.
 */
    else
    {
        char name[DITS_C_NAMELEN];
        DitsTransIdType transid;
        DitsPathType path;
        DitsGetEntInfo(sizeof(name),name,&path,&transid,&reason,
                            &reasonstat,status);
        if (path == teaPath)
            teaTransid = 0;
        else
            coffeeTransid = 0;
/*
 *      If necessary, wait for the next one, otherwise stage to the
 *      next part (CTestStartActions)
 */
        if ((teaTransid == 0)&&(coffeeTransid == 0))
        {
            DitsPutHandler(CTestStartActions,status);
            DitsPutRequest(DITS_REQ_STAGE,status);
        }
        else
        {
            if (TIMEOUT > 0)
                DitsPutDelay(&timeout,status);
            DitsPutRequest(DITS_REQ_MESSAGE,status);
        }
    }
}
```

This is how the same code could be rewritten using **GitPathGetInit()** and **GitPathGetComp()**. The problems are that you cannot have more then one GetPath's operation outstanding at the same time, you can't use this technique in user interface code and you have a couple of extra action stages to get the job done. But it's a lot simpler.

```
static void CTestFindPaths(StatusType *status)
{
    if (*status != STATUS__OK) return;
/*
 *  Initiate getting the path to the tea task.  CTestTeaFound will
 * be invoked when we have a result.
 */
    GitPathGetInit("TEA",MESSAGEBYTES,MAXMESSAGES,REPLYBYTES,MAXREPLIES,
      TIMEOUT,CTestTeaFound,0,status);

}

static void CTestTeaFound(StatusType *status)
{
    if (*status != STATUS__OK) return;
/*
 *  Complete the path get operation and get the details.
 */
    GitPathGetComp(&teaPath,0,status);
    if (*status != STATUS__OK)
    {
        ErsRep(0,status,"Failed to get path to task  TEA: %s",
                                      DitsErrorText(*status));
    }
    else
/*
 * Initiate getting the path to the coffee task.  CTestCoffeeFound will
 * be invoked when we have a result.
 */
        GitPathGetInit("COFFEE",MESSAGEBYTES,MAXMESSAGES,REPLYBYTES,
                      MAXREPLIES,TIMEOUT,CTestCoffeeFound,0,status);
}
static void CTestCoffeeFound(StatusType *status)
{
    if (*status != STATUS__OK) return;
/*
 *  Complete the path get operation and get the details.
 */
    GitPathGetComp(&coffeePath,0,status);
    if (*status != STATUS__OK)
    {
        ErsRep(0,status,"Failed to get path to task  COFFEE: %s",
                                         DitsErrorText(*status));
    }
    else
    {
/*
 * Stage to the routine that does the real work.
 */
        DitsPutHandler(CTestStartActions,status);
        DitsPutRequest(DITS_REQ_STAGE,status);
    }
}
```

9.5 GitTpi routines

Modules written using an object oriented style will normally require internal storage to maintain state information. This is no problem on VMS an UNIX machines, where each **DRAMA** task runs in its own address space. But, under some operating systems, e.g. VxWorks, all tasks run in the same address space. Here, `static` and `extern` variables are common to all tasks. The **DitsPutUserData()** and **DitsGetUserData()** (see [3]) routines provide a way around this problem. Unfortunately, if used in a simple way, a task must know about all the packages using these routines. It would be nice if all packages could have transparent access to this storage area. The **GitTpi** routines provide this. The main module of your program should call **GitTpiInit()**. Then, each package can call **GitTpiPut()** to store data for which it would otherwise call **DitsPutUserData()**. The packages can then call **GitTpiGet()** to retrieve the data. For this to work, only the **GitTpi** routines should call **DitsPutUserData()** and **DitsGetUserData()**.

9.6 Parameter Monitor Support

Dits provides support for one task to monitor the values of parameters in other tasks. As the low level interface to parameter monitoring provided is a bit complex, **Git** provides a simpler interface.

GitMonitorMessage() can be used to initiate monitor messages of any type. It wraps up the process of creating arguments structures from lists of parameters.

The routines **GitMonitorStart()** and **GitMonitorForward()** are used to initiate monitoring transactions in which most of the work is done by the **Git** package. You need only supply callback routines. The former routine is very usefull if you want parameters in one task to reflect the values of parameters in another task.

9.6.1 Procedure Types

Various procedures are passed the **GitMonitorStart()** and **GitMonitorForward()** routines. 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 void  (*GitMonitorStartedType)(
                int id,             /* (>) Id of the monitor transaction */
                void * client_data,
                StatusType *status);
  ```

-
  ```
  typedef void  (*GitMonitorChangedType)(
                char * name,        /* (>) Name of parameter            */
                SdsCodeType type,   /* (>) Sds type of parameter        */
                void * value,       /* (>) Value of parameter           */
                void * client_data,
                StatusType *status);
  ```

  Value is the address of any item of the type indicated by the Sds code. Value is not valid for Array or Structured items. In these cases it is set too zero. A special case is character arrays, which are treated as strings. In this case, the type will be ARG_STRING instead of SDS_CHAR.

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

References

[1] Tony Farrell, AAO. *05-Aug-1993, Guide to writing Drama tasks.* Anglo-Australian Observatory **DRAMA** Software Document 3.

[2] Tony Farrell, AAO. *01-Apr-1993,* **DRAMA** *Error reporting System.* Anglo-Australian Observatory **DRAMA** Software Document 4.

[3] Tony Farrell, AAO. *05-Jan-1994, Distributed Instrumentation Tasking System.* Anglo-Australian Observatory **DRAMA** Software Document 5.

[4] Tony Farrell, AAO. *18-Feb-1992, A portable Message Code System.* Anglo-Australian Observatory **DRAMA** Software Document 6.

[5] Jeremy Bailey , AAO. *9-Sep-1992, Self-defining Data System.* Anglo-Australian Observatory.

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

A Git Routines

This appendix describes the routines provided by the Git package.

A.1 GitActivate — Activate the Generic instrument task action handlers.

Function: Activate the Generic instrument task action handlers.

Description: Default actions handlers for all the standard Generic instrument task actions are put using DitsPutActionHandlers and the standard parameters are created using SdpCreate.

Handlers are added for the following actions

`INITIALISE`	Simply set simulation and output a message,
`CTRLC`	Does nothing.
`DUMP_LOG`	Currently, does nothing.
`EXIT`	Causes task to exit.
`LOG_LEVEL`	Currently does nothing
`POLL`	Outputs a message and waits for a message.
`POLL_PARAMETER`	Currently does nothing
`RESET`	Output a message.
`SIMULATE_LEVEL`	Sets the simulation level.
`UPDATE_NBD`	Does nothing.
`UMONITOR`	Calls the DmonCommand routine.

The following parameters are created -

`LOG_LEVEL`	Default value “`NONE`”
`SIMULATE_LEVEL`	No default value
`TIME_BASE`	Default value 1.0
`ENQ_DEV_TYPE`	Default value “`IDT`”
`ENQ_DEV_DESCR`	Default value “Generic Instrument task”
`ENQ_VER_NUM`	Default value = git version number
`ENQ_VER_DATE`	Default value = git version date
`ENQ_DEV_NUMTIEM`	Default value 0.

Language: C

Call:

(Void) = GitActivate (parsysid, status)

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

(>) parsysid (SdsIdType) The parameter system id returned by a call to SdpInit

(!) status (StatusType *) Modified status.

Include files: Git.h

External functions used:

DitsPutActionHandlers	Dits	Put some action handlers.
SdpCreate	Dits-Sdp	Create parameters.
MessPutFacility	Mess	Add a new message facility.

External values used: none

Prior requirements: DitsInit and SdpInit should have been called. Should not be called from a Dits action handler routine.

Support: Tony Farrell, AAO

A.2 GitArgGetD — Gets a double floating point argument value with checking against a range.

Function: Gets a double floating point argument value with checking against a range.

Description: Assumes "id" is an Sds Structure item containing the required argument and "name" if the name of the argument.

If "range" is non-zero, it is an array with two entries - the lower and upper limits of the allowable range of actValue.

If there is any error in getting the value, including its value not being in range, the actual value is set to that specified by "defVal" and status will be reset to `OK`, (see flags arguments to change the effect on status).

Language: C

Call:

(void) = GitArgGetD (id, name, position, range, defVal, flags, actValue, status)

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

- (>) **id (SdsIdType)** Id of the argument system.
- (>) **name (Char *)** Name of the argument to get
- (>) **position (int)** The position of the argument. This is used if "name" is a null pointer, an empty string or if we can't find an `SDS` structure item of that name.
- (>) **range (Double [])** If non-zero, array of two values, the lower and upper limit for value.
- (>) **defVal (Double)** The Default value. Note, this is not validated against the range.
- (>) **flags (Int)** A bit-mask of flags. Possible values are

`GIT_M_ARG_KEEPERR`	If we use the default value, return the error which caused it to be used.
`GIT_M_ARG_KEEPVALERR`	Only retain the error if we found the item (and its value was in error). If we did not find the item, then return status ok and the default.

- (<) **actValue (Double *)** The actual value is written here.
- (!) **status (StatusType *)** Modified status. If we have an error and `GIT_M_ARG_KEEP-ERR` is true then a message is reported using `ErsRep` and status is set. In addition to error codes from the underlying Arg and Sds routines, one of the following may be returned.

`GIT__NOARG`	The argument id is zero or no item was found. Will not be returned if `GIT_M_ARG_KEEPVALERR` flag is set, in that case, just return the default with status ok.
`GIT__ARGLTMIN`	Argument value less than range[0].
`GIT__ARGGTMAX`	Argument value greater than range[1].

Include files: Git.h

External functions used:

SdsFreeId	Sds	Free an Sds id
GitArgNamePos	Git	Get an argument Id by name or position
ArgCvt	Arg	Convert an id to a specified type.
ErsRep	Ers	Report an error.
ErsPush	Ers	Increase error context
ErsPop	Ers	Decreate error context
ErsAnnul	Ers	Annul error messages

External values used: none

Prior requirements: As above.

Support: Tony Farrell, `AAO`

A.3 GitArgGetI — Gets an integer argument value with checking against a range.

Function: Gets an integer argument value with checking against a range.

Description: Assumes “id” is an Sds Structure item containing the required argument and “name” if the name of the argument.

If “range” is non-zero, it is an array with two entries - the lower and upper limits of the allowable range of actValue.

If there is any error in getting the value, including its value not being in range, the actual value is set to that specified by “defVal” and status will be reset to `OK`. (see flags arguments to change the effect on status).

Language: C

Call:

(void) = GitArgGetI (id, name, position, range, defVal, flags, actValue, status)

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

(>) **id (SdsIdType)** Id of the argument system.

(>) **name (Char *)** Name of the argument to get

(>) **position (int)** The position of the argument. This is used if “name” is a null pointer, an empty string or if we can’t find an `SDS` structure item of that name.

(>) **range (Long Int [])** If non-null, array of two values, the lower and upper limit for value.

(>) **defVal (Long Int)** The Default value. Note, this is `NOT` validated against the range.

(>) **flags (Int)** A bit-mask of flags. Possible values are

`GIT_M_ARG_KEEPERR`	If we use the default value, return the error which caused it to be used.
`GIT_M_ARG_KEEPVALERR`	Only retain the error if we found the item (and its value was in error). If we did not find the item, then return status ok and the default.

(<) **actValue (Long int *)** The actual value is written here.

(!) **status (StatusType *)** Modified status. If we have an error and `GIT_M_ARG_KEEP-ERR` is true then a message is reported using `ErsRep` and status is set. In addition to error codes from the underlying Arg and Sds routines, one of the following may be returned

`GIT__NOARG`	The argument id is zero or no item was found. Will not be returned if `GIT_M_ARG_KEEPVALERR` flag is set, in that case, just return the default with status ok.

`GIT__ARGLTMIN`	Argument value less than range[0].
`GIT__ARGGTMAX`	Argument value greater than range[1].

Include files: Git.h

External functions used:

SdsFreeId	Sds	Free an sds id
GitArgNamePos	Git	Get an argument Id by name or position
ArgCvt	Arg	Convert an id to a specified type.
ErsRep	Ers	Report an error.
ErsPush	Ers	Increase error context
ErsPop	Ers	Decreate error context
ErsAnnul	Ers	Annul error messages

External values used: none

Prior requirements: As above

Support: Tony Farrell, AAO

A.4 GitArgGetL — Gets a logical argument value.

Function: Gets a logical argument value.

Description: Assumes "id" is an Sds Structure item containing the required argument and "name" if the name of the argument.

This routine fetches the value of a logical argument. There are several possibilties.

1. The argument value can be converted to an integer and if non zero, it is considered true, otherwise false.
2. The argument value can be converted to an integer and the `LSB` inidicates if the value is true or false.
3. An array of strings is provided. The array contains pairs of `TRUE`/`FALSE` strings. For example, "`TRUE`" and "`FALSE`" themselves or "`OPEN`" and "`SHUT`" etc.

The value is first treated as an integer, the selection of technique 1 or 2 is based on a flag. If this conversion fails and the strings argument is provided, then technique three is tried.

If there is any error in getting the value, the actual value is set to that specified by "defVal" and status will be reset to `OK`, (see flags arguments to changes the effect on status).

Language: C

Call:
(void) = GitArgGetL (id, name, position, strings, defVal, flags, actValue, status)

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

(>) **id (SdsIdType)** Id of the argument system.

(>) **name (Char *)** Name of the argument to get

(>) **position (int)** The position of the argument. This is used if "name" is a null pointer, an empty string or if we can't find an `SDS` structure item of that name.

(>) **strings (GitLogStrType [])** If nonzero, then this is an array describing the possible values for `TRUE` and `FALSE`. "strings[n].true" is a pointer to a character string value representing `TRUE`, while "strings[n].false" is a pointer character string represeting `FALSE`. The maximum length is `GIT_ARG_LMAX` characters (20).

An entry containing two null pointers indicates the end of the array.

If zero, the argument should be able to be converted to an integer value.

(>) **defVal (Int)** The Default value.

(>) **flags (Int)** A bit-mask of flags. Possible values are

`GIT_M_ARG_UPPER`	Convert value to upper case.
`GIT_M_ARG_LOWER`	Convert value to lower case.
`GIT_M_ARG_LASTBIT`	If set, we only consider the `LSB` of integer values. If not set, only a value of 0 is false.
`GIT_M_ARG_KEEPERR`	If we use the default value, return the error which caused it to be tasken.
`GIT_M_ARG_ABBREV`	Allow abbreviations. The minimum abbreviation is 2 characters. The first value in the "strings" array which is correct to length of the supplied value is used.
`GIT_M_ARG_KEEPVALERR`	Only retain the error if we found the item (and its value was in error). If we did not find the item, then return status ok and the default.

(<) actValue (int *) The actual value is written here. Set to 1 for true and 0 for false.

(!) status (StatusType *) Modified status. If we have an error and `GIT_M_ARG_KEEPERR` is true then a message is reported using `ErsRep` and status is set. In addition to error codes from the underlying Arg and Sds routines, one of the following may be returned

`GIT__NOARG`	The argument id is zero or no item was found. Will not be returned if `GIT_M_ARG_KEEPVALERR` flag is set, in that case, just return the default with status ok.
`GIT__ARGVAL`	Invalid argument value, not one of those specified in the strings array.

Include files: Git.h

External functions used:

SdsFreeId	Sds	Free an sds id
GitArgNamePos	Git	Get an argument Id by name or position
ArgCvt	Arg	Convert an id to a specified type.
ErsRep	Ers	Report an error message
ErsPush	Ers	Increase error context
ErsPop	Ers	Decreate error context
ErsAnnul	Ers	Annul error messages
islower	`CRTL`	Determines if a character is lower case.
isupper	`CRTL`	Determines if a character is upper case.
toupper	`CRTL`	Convert a character to upper case.
tolower	`CRTL`	Convert a character to lower case.
strcmp	`CRTL`	Compare one string to anoterh
strncpy	`CRTL`	Copy one string to another.

External values used: none

Prior requirements: As above

Support: Tony Farrell, `AAO`

A.5 GitArgGetS — Gets a string argument value with checking against acceptable values.

Function: Gets a string argument value with checking against acceptable values.

Description: Assumes "id" is an Sds Structure item containing the required argument and "name" if the name of the argument.

If "values" is non-zero, then it is an array of acceptables values. If the actual value is not in this array, then status is set to `GIT__ARGVAL`.

If "defVal" is non-zero, then if there is any error in getting the value, including its value not being acceptable, the actual value is it to that specified by "defVal" and status will be reset to `OK`, (see flags arguments to change the effect on status).

Language: C

Call:

(void) = GitArgGetS (id, name, position, values, defVal, flags, actValLen, actValue, index, status)

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

- **(>) id (SdsIdType)** Id of the argument system.
- **(>) name (Char *)** Name of the argument to get
- **(>) position (int)** The position of the argument. This is used if "name" is a null pointer, an empty string or if we can't find an `SDS` structure item of that name.
- **(>) values (Char *[])** If non-null, array of pointers to the possible values. Terminate with 0.
- **(>) defVal (Char *)** If non-null, the default value. Note, this value is `NOT` validated against the values array.
- **(>) flags (Int)** A bit-mask of flags. Possible values are

`GIT_M_ARG_UPPER`	Convert value to upper case.
`GIT_M_ARG_LOWER`	Convert value to lower case.
`GIT_M_ARG_KEEPERR`	If we use the default value, return the error which caused it to be taken. (If we don't have a default value, then we return the error).
`GIT_M_ARG_ABBREV`	Allow abbreviations. The minimum abbreviation is 2 characters. The first value in the "values" array which is correct to length of the supplied value is used. The full length value is copyied from "values" to actValue.
`GIT_M_ARG_KEEPVALERR`	Only retain the error if we found the item (and its value was in error). If we did not find the item, then return status ok and the default.

(>) actValLen (Int) Length of actValue

(<) actValue (Char *) The actual value is written here.

(<) index (Int *) If non-null - if we found a value in the values array, this is the index to that value. Otherwise, it is set to -1.

(!) status (StatusType *) Modified status. If we have an error and `GIT_M_ARG_KEEPERR` is true then a message is reported using `ErsRep` and status is set. In addition to error codes from the underlying Arg and Sds routines, one of the following may be returned

`GIT__NOARG`	The argument id is zero or no item was found. Will not be returned if `GIT_M_ARG_KEEPVALERR` flag is set, in that case, just return the default with status ok.
`GIT__ARGVAL`	Invalid argument value, not one of those specified in the values array.

Include files: Git.h

External functions used:

SdsFreeId	Sds	Free and Sds Id.
ArgGetString	Arg	Get an argument value
ErsRep	Ers	Report an error.
ErsPush	Ers	Increase error context
ErsPop	Ers	Decreate error context
ErsAnnul	Ers	Annul error messages
islower	`CRTL`	Determines if a character is lower case.
isupper	`CRTL`	Determines if a character is upper case.
toupper	`CRTL`	Convert a character to upper case.
tolower	`CRTL`	Convert a character to lower case.
strcmp	`CRTL`	Compare one string to anoterh
strncpy	`CRTL`	Copy one string to another.

External values used: none

Prior requirements: As above

Support: Tony Farrell, `AAO`

A.6 GitArgGetStruct — Gets a structure argument value.

Function: Gets a structure argument value.

Description: This routine returns the id of an Sds structure which may be specified either directly as the argument to an action or as the name of an SDS file containing the structure.

If the argument system id is non-zero, points to an Sds structure and the structure name is not “ArgStructure”, then the id of this structure is returned.

If the argument system id is non-zero and points to the an Sds item named “ArgStructure”, the we try to find (by name or position), the requested item. If such an item is found and it is a Sds structure, the id of the item is returned. Otherwise, we interpet the value of the item an a filename and try to read the file. If the succeeds, then we return the id of the structure read from the file use SdsRead.

If we still don’t have a value and defValue is non-zero, then it is treated as the name of a file which we read using SdsRead. The resulting Sds id is returned.

If we read the value from a file, the name of the file is returned in actName. Otherwise, the name of the structure is returned in actName.

Language: C

Call:
(void) = GitArgGetStruct (id, name, position, defVal, actNameLen, actName, actValue status)

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

- (>) **id (SdsIdType)** Id of the argument system.
- (>) **name (Char *)** Name of the argument to get
- (>) **position (int)** The position of the argument. This is used if “name” is a null pointer, an empty string or if we can’t find an SDS structure item of that name.
- (>) **defVal (Char *)** If non-null, the default value. We try to read a file of this name.
- (>) **actNameLen (int)** Length of buffer pointed to by actName.
- (<) **actName (Char *)** If we read a file, the name of the file is written here. Otherwise, the name of the structure is written here.
- (<) **actValue (SdsIdType *)** The actual value is written here.
- (!) **status (StatusType *)** Modified status. If we have an error and no default was supplied, then a message is reported using `ErsRep` and status is set. In addition to error codes from the underlying Arg and Sds routines, one of the following may be returned

`GIT__NOARG`	The argument id is zero.
`GIT__ARGVAL`	The argument value is invalid in some way.
`SDS__FOPEN`	We tried to read a file but file. Note that in this case actName will have the name of the file that we tried to open.

Include files: Git.h

External functions used:

ArgGetString	Arg	Get an argument value
ErsRep	Ers	Report an error.
ErsPush	Ers	Increase error context
ErsPop	Ers	Decreate error context
ErsAnnul	Ers	Annul error messages
strcmp	CRTL	Compare one string to another
strncpy	CRTL	Copy one string to another.
SdsRead	Sds	Read an sds item from a file
SdsInfo	Sds	Obtain details of an Sds item.

External values used: none

Prior requirements: As above

Support: Tony Farrell, AAO

A.7 GitArgNamePos — Return the Id of an argument given its name and position.

Function: Return the Id of an argument given its name and position.

Description: Returns an Sds Id of the argument of the given name within the specified argument system. If there is no argument of that name, or the name is specified as null, return the argument in the specified position.

DEPRECATED: Prefer GitArgNamePos2().

Language: C

Call:

(void) = GitArgNamePos (id, name, position, argid , status)

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

- (>) **id (SdsIdType)** Id of the argument system.
- (>) **name (Char *)** Name of the argument to get
- (>) **position (Int)** Position of argument, first argument is 1. This is used if "name" is a null pointer, an empty string or if we can't find an SDS structure item of that name.
- (<) **argid (SdsIdType *)** Id of argument.
- (!) **status (StatusType *)** Modified status.

Include files: Git.h

External functions used:

SdsFind	SDS	Find an item by name
SdsIndex	Sds	Find an item by position

External values used: none

Prior requirements:

Support: Tony Farrell, AAO

A.8 GitEnvGetS — Get a value of a logical name/environment variable

Function: Get a value of a logical name/environment variable

Description: Under VMS, translate the name of the given logical name/Cli symbol. Under UNIX/VxWorks, return the value of the given environment variable. Under VxWorks version 5.0 and before, return false.

This routine is different for each operation system. Under VxWorks 5.0 and before, it always returns false, as there is no equivalent to logical names/environment variables in that environment.

Language: C

Call:
(int) = GitEnvGetS (Name, valLen, Val, status)

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

(>) eName (Char *) Logical name(VMS)/Environment Variable name (UNIX).

(>) valLen (int) The length allowed for the value

(<) val (Char *) The actual value.

(!) status (StatusType *) Modified status.

Include files: Git.h

Function value: Returns true for a successfull translation. False otherwise.

External functions used:

DitsGetSymbol	DITS	Get the value of an environement variable.

External values used: none

Prior requirements: none

Support: Tony Farrell, AAO

A.9 GitMonitorForward — Setup and run a Monitor Forward operation.

Function: Setup and run a Monitor Forward operation.

Description: A monitor start message is sent to the task whose path is specified, with the optional list of parameter supplied. Responses are handled and user supplied routines invoked when necessary.

This routine is intended to hide the details of parameter monitoring. It will set its' own Obey and Kick handlers to handle the results. It also uses DitsPutActData to store information.

On return from this routine, the caller should call DitsPutRequest with a request of `DITS_REQ_MESSAGE` and return to the fixed part.

User routines supplied below will be invoked when corresponding messages arrive. If the user does not supply a Completed routine, then the action which invoked this routine will exit when the monitor message terminates.

Language: C

Call:

(void) = GitMonitorForward (path, task, action, Started, Completed, Unexpected, client_data, count, status, [parameters,[...]])

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

- **(>) path (DitsPathType)** Path to the task in which parameters are to be monitored.
- **(>) task (char *)** Task to which to send the monitor message.
- **(>) action (char *)** Name of the action in that task to obey.
- **(>) Started (GitMonitorStartedType)** Invoked when a monitor started message is received. Optional. If not required, specify 0. On return from this routine, a request of `DITS_REQ_MESSAGE` is put.
- **(>) Completed (GitMonitorResponseType)** Invoked when monitoring completes for whatever reason. You can use the normal Dits calls to get more information. No request is put on return from this routine so the default is for the action to complete when monitoring completes. The status returned by this routine will be the completion status of the action. DitsPutActData() is used to restore the original value of ActData before this routine is invoked. If this routine is not supplied then the action which invokes this routine will complete when monitoring completes and the completion status of the monitoring will be the completion status of the action.
- **(>) Unexpected (GitMonitorResponseType)** Invoked for unexpected messages received while monitoring. These would normally be as the result of transactions started by the user. Normal Dits routines can be used to get details. If not supplied, the a message is output. A request of `DITS_REQ_MESSAGE` is put on return from this routine.
- **(>) client_data (void *)** Client data available to the handler routine.
- **(>) count (int)** Number of parameters supplied

- **(!) status (StatusType *)** Modified status.
- **(>) parameters (char *)** A list of parameters to monitor.

Include files: Git.h

External functions used:

ArgNew	Arg	Create an argument structure.
ArgPutString	Arg	Put a string item into an argument structure.
DitsNameSet	Dits	Set a name item
DitsInitiateMessage	Dits	Send a message to a task.
DitsPutObeyHandler	Dits	Put an obey message handler
DitsPutKickHandler	Dits	Put a Kick message handler
DitsPutActData	Dits	Save data.
sprintf	CRTL	Formated print into a string.

External values used: none

Prior requirements: Should only be called from a Dits application action routine.

Support: Tony Farrell, AAO

A.10 GitMonitorMessage — Initiate a Monitor Message.

Function: Initiate a Monitor Message.

Description: This routine calls DitsInitiateMessage to send a monitor message of type with the specified list of parameters. This is a simpilied interface to DitsInitiateMessage for monitor message. It is up to the caller to handle the results.

Language: C

Call:
(void) = GitMonitorMessage (id, path, type, count, transid, status, [parameters,[...]])

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

- **(>) id (int)** Monitor id returned by a Monitor Started message. Ignored for `START` and `FORWARD` type messages.
- **(>) path (DitsPathType)** Path to the task in which parameters are being monitored.
- **(>) type (GitMonitorMsgType)** One of `GIT_MON_START`, `GIT_MON_FORWARD`, `GIT_MON_ADD`, `GIT_MON_DELETE` or `GIT_MON_CANCEL`.
- **(>) count (int)** Number of parameters supplied
- **(<) transid (DitsTransIdType)** Id of the resulting transaction.
- **(!) status (StatusType *)** Modified status.
- **(>) parameters (char *)** A list of parameters to monitor. For a `FORWARD` type message, the first two paramters should be task name and action name to which parameter change messages are to be forwarded.

Include files: Git.h

External functions used:

ArgNew	Arg	Create an argument structure.
ArgPuti	Arg	Put an integer item into an argument structure.
ArgPutString	Arg	Put a string item into an argument structure.
DitsNameSet	Dits	Set a name item
DitsInitiateMessage	Dits	Send a message to a task.
sprintf	CRTL	Formated print into a string.

External values used: none

Prior requirements: Should only be called from a Dits application routine or when context is `DITS_CTX_UFACE`.

Support: Tony Farrell, `AAO`

A.11 GitMonitorStart — Setup and run a monitor transaction.

Function: Setup and run a monitor transaction.

Description: A monitor start message is sent to the task whose path is specified, with the optional list of parameter supplied. Responses are handled and user supplied routines invoked when necessary.

This routine is intended to hide the details of parameter monitoring. It will set its own Obey and Kick handlers to handle the results. It also uses DitsPutActData to store information.

On return from this routine, the caller should call DitsPutRequest with a request of `DIT-S_REQ_MESSAGE` and return to the fixed part.

User routines supplied below will be invoked when corresponding messages arrive. If the user does not supply a Completed routine, then the action which invoked this routine will exit when the monitor message terminates.

Language: C

Call:
(void) = GitMonitorStart (path, Started, Changed, Completed, Unexpected, client_data, count, status, [parameters,[...]])

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

(`>`) path (DitsPathType) Path to the task in which parameters are to be monitored.

(`>`) Started (GitMonitorStartedType) Invoked when a monitor started message is received. Optional. If not required, specify 0. On return from this routine, a request of `DITS_REQ_MESSAGE` is put.

(`>`) Changed (GitMonitorChangedType) Invoked when a parameter changed monitor message is received. If the parameter whose value has changed is a non-structured, non-array item or a character string, its name, value and Sds type will be supplied to this routine. For arrays or structures, just the name and type (value set to 0). You can use

() DitsGetArgument () to get the actual sds item id. On return from this routine, a request of `DITS_REQ_MESSAGE` is put.

(`>`) Completed (GitMonitorResponseType) Invoked when monitoring completes for whatever reason. You can use the normal Dits calls to get more information. No request is put on return from this routine so the default is for the action to complete when monitoring completes. The status returned by this routine will be the completion status of the action. DitsPutActData() is used to restore the original value of ActData before this routine is invoked. If this routine is not supplied then the action which invokes this routine will complete when monitoring completes and the completion status of the monitoring will be the completion status of the action.

(`>`) Unexpected (GitMonitorResponseType) Invoked for unexpected messages received while monitoring. These would normally be as the result of transactions started by the user. Normal Dits routines can be used to get details. If not supplied, the a message is output. A request of `DITS_REQ_MESSAGE` is put on return from this routine.

- (>) **client_data (void *)** Client data available to the handler routine.
- (>) **count (int)** Number of parameters supplied
- (!) **status (StatusType *)** Modified status.
- (>) **parameters (char *)** A list of parameters to monitor.

Include files: Git.h

External functions used:

ArgNew	Arg	Create an argument structure.
ArgPutString	Arg	Put a string item into an argument structure.
DitsNameSet	Dits	Set a name item
DitsInitiateMessage	Dits	Send a message to a task.
DitsPutObeyHandler	Dits	Put an obey message handler
DitsPutKickHandler	Dits	Put a Kick message handler
DitsPutActData	Dits	Save ActData item.
DitsGetActData	Dits	Get ActData item.
sprintf	`CRTL`	Formated print into a string.

External values used: none

Prior requirements: Should only be called from a Dits application action routine.

Support: Tony Farrell, `AAO`

A.12 GitMonitorStartF — Setup and run a monitor transaction. Version with flags specifiable.

Function: Setup and run a monitor transaction. Version with flags specifiable.

Description: A monitor start message is sent to the task whose path is specified, with the optional list of parameter supplied. Responses are handled and user supplied routines invoked when necessary.

This routine is intended to hide the details of parameter monitoring. It will set its own Obey and Kick handlers to handle the results. It also uses DitsPutActData to store information.

On return from this routine, the caller should call DitsPutRequest with a request of `DIT-S_REQ_MESSAGE` and return to the fixed part.

User routines supplied below will be invoked when corresponding messages arrive. If the user does not supply a Completed routine, then the action which invoked this routine will exit when the monitor message terminates.

This version of GitMonitorStart() allows you to specify the message flags.

Language: C

Call:
(void) = GitMonitorStartF (path, Flags, Started, Changed, Completed, Unexpected, client_data, count, status, [parameters,[...]])

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

(>) **path (DitsPathType)** Path to the task in which parameters are to be monitored.

(>) **Flags (int)** Flags to the monitor messages. Set `DITS_M_SENDCUR` if 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.

(>) **Started (GitMonitorStartedType)** Invoked when a monitor started message is received. Optional. If not required, specify 0. On return from this routine, a request of `DITS_REQ_MESSAGE` is put.

(>) **Changed (GitMonitorChangedType)** Invoked when a parameter changed monitor message is received. If the parameter whose value has changed is a non-structured, non-array item or a character string, its name, value and Sds type will be supplied to this routine. For arrays or structures, just the name and type (value set to 0). You can use

() **DitsGetArgument ()** to get the actual sds item id. On return from this routine, a request of `DITS_REQ_MESSAGE` is put.

(>) **Completed (GitMonitorResponseType)** Invoked when monitoring completes for whatever reason. You can use the normal Dits calls to get more information. No request is put on return from this routine so the default is for the action to complete when monitoring completes. The status returned by this routine will be the completion status of the action. DitsPutActData() is used to restore the original value of ActData before this routine is invoked. If this routine is not supplied then the action which invokes this routine will complete when monitoring completes and the completion status of the monitoring will be the completion status of the action.

- (>) **Unexpected (GitMonitorResponseType)** Invoked for unexpected messages received while monitoring. These would normally be as the result of transactions started by the user. Normal Dits routines can be used to get details. If not supplied, the a message is output. A request of `DITS_REQ_MESSAGE` is put on return from this routine.
- (>) **client_data (void *)** Client data available to the handler routine.
- (>) **count (int)** Number of parameters supplied
- (!) **status (StatusType *)** Modified status.
- (>) **parameters (char *)** A list of parameters to monitor.

Include files: Git.h

External functions used:

ArgNew	Arg	Create an argument structure.
ArgPutString	Arg	Put a string item into an argument structure.
DitsNameSet	Dits	Set a name item
DitsInitiateMessage	Dits	Send a message to a task.
DitsPutObeyHandler	Dits	Put an obey message handler
DitsPutKickHandler	Dits	Put a Kick message handler
DitsPutActData	Dits	Save ActData item.
DitsGetActData	Dits	Get ActData item.
sprintf	`CRTL`	Formated print into a string.

External values used: none

Prior requirements: Should only be called from a Dits application action routine.

Support: Tony Farrell, `AAO`

A.13 GitParEnvGetS — Get a value from a parameter or, failing that, the environment.

Function: Get a value from a parameter or, failing that, the environment.

Description: Firstly, try and obtain a value for the parameter using the parameter system. If this fails or returns the specified bad value, try translating the supplied logical name (VMS) or environment variable(UNIX/VxWorks). IF this fails or returns the specified bad value, return the specified default.

Note that prior to VxWorks 5.1, Vxworks did not support environement variables.

Language: C

Call:
(Void) = GitParEnvGetS (pName, eName, badVal, defVal, actValLen, actVal, status)

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

- (>) **pName (Char *)** Parameter name (null terminated)
- (>) **eName (Char *)** Logical name(VMS)/Environment Variable (UNIX/VxWorks). This is used if the above parameter cannot be read or if it has the specified invalid value (null terminated). If zero, then only the parameter is used.
- (>) **badVal (Char *)** The bad value that causes the logical name of the default value to be used
- (>) **defVal (Char *)** Default value to be returned if no valid value can be obtained.
- (>) **actValLen (int)** The length allowed for the actual value
- (<) **actVal (Char *)** The actual value.
- (!) **status (StatusType *)** Modified status.

Include files: Git.h

External functions used:

GitEnvGetS	Git	The the value of an environment variable/logical name
strcmp	CRTL	Compare two strings
strncpy	CRTL	Copy one string to another
SdpGetString	Sdp	Get the value of a parameter

External values used: none

Prior requirements: Must only be called as part of a Dits application routine. Assumes a parameter system is enabled and ArgGetString can be used to get the parameter value, given the parameter system id. (This is case if the task has called GitActivate)

Support: Tony Farrell, AAO

A.14 GitPathGetComp — Complete the getting of a path.

Function: Complete the getting of a path.

Description: Should be called by the action routine specified by GitPathGetInit. This routine returns the actual path and the status of the operation.

This value stored by DitsPutActData() is returned to its value before the call to GitPathGetInit().

Language: C

Call:
(void) = GitPathGetComp (path, client_data, status)

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

(<) path (DitsPathType *) If status is ok, the path to the task.

(<) client_data (void **) Client data supplied to GitPathGetInit. If an address of zero is supplied, nothing is returned.

(!) status (StatusType *) Modified status. If status is ok, then is will be set to the status of the GetPath operation. In addition to values from the underlying systems, the following codes are possible

`GIT__PATH_TIMEOUT`	The user's timeout expired before the path was found.
`GIT__PATH_INV_ENTRY`	Unexpected action entry reason.

Include files: Git.h

External functions used:

DitsGetActData	Dits	Get data assocaited with action
DitsPutActData	Dits	Store data assocaited with action
free	`CRTL`	Deallocate memory

External values used: none

Prior requirements: Should only be called from an action routine specified to a corresponding call to GitPathGetInit().

Support: Tony Farrell, `AAO`

A.15 GitPathGetInit — Initiate the getting of a path.

Function: Initiate the getting of a path.

Description: This routine is a wrap around for the DitsGetPath routine, simpilfying its use.

The application firsts calls this routine, specifing the relavent details for DitsGetPath and a handler routine.

Your handler routine will be invoked when either the path is found or an error occurs. The handler is invoked by calling DitsPutHandler() with its address and then calling DitsPutRequest() with a request of `DITS_REQ_STAGE`. This ensures consistency in the interface.

The handler `MUST` then call DitsPathGetComp() to tidy up and retrieve details of the path.

This routine calls DitsPutHandler() and DitsPutRequest(). The calling action routine should return immediately after calling this routine to await the response.

This routine uses the DitsPutActData() routine to store information. Any value that was stored with DitsPutActData() before entry to this routine will be restored by GitPathGetComp().

A Kick handler is enabled which will respond to kicks by canceling the GetPath operation can causing the action to exit. If any other response is required, the user can enable his own kick handler after the return from this routine. When such a kick handler is invoke, it should call GitPathGetComp to tidy up, but note that the path and status returned may not be valid.

Language: C

Call:

(void) = GitPathGetInit (name, messageBytes, maxMessages, replyBytes, maxReplies , handler, client_data, status)

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

- (>) **name (char *)** As per DitsGetPath().
- (>) **messageBytes (int)** As per DitsGetPath().
- (>) **maxMessages (int)** As per DitsGetPath().
- (>) **replyBytes (int)** As per DitsGetPath().
- (>) **maxReplies (int)** As per DitsGetPath().
- (>) **timeout (int)** Timeout in seconds. (<= 0 no timeout). If we do not get the path in the specified time, an error is returned.
- (>) **handler (DitsActionRoutineType)** Called at completion of the get path operation a a Dits action routine.
- (>) **client_data (void *)** Client data available to the handler routine.
- (!) **status (StatusType *)** Modified status.

Include files: Git.h

External functions used:

DitsGetActData	Dits	Get data assocaited with action
DitsPutActData	Dits	Store data assocaited with action
DitsGetReason	Dits	Get action entry details.
DitsPutHandler	Dits	Put a new action handler
DitsPutRequest	Dits	Put a request.
DitsGetPath	Dits	Initiate Getting of a path.
DitsDeltaTime	Dits	Create a delta time value
DitsPutDelay	Dits	Put an action timeout
malloc	`CRTL`	Allocate memory

External values used: none

Prior requirements: Should only be called from a User action routine. Cannot be used with in Uface context.

Support: Tony Farrell, `AAO`

A.16 GitPutDelay — Set Action delay or timeout.

Function: Set Action delay or timeout.

Description: This routine is a simplfied interface to the common sequences DitsDeltaTime - DitsPutDelay.

Language: C

Call:
(Void) = GitPutDelay (delay, status)

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

(>) delay (double) The delay before rescheduling.

(!) status (StatusType *) Modified status.

Include files: Git.h

External functions used:

DitsDeltaTime	Dits	Put a delta time in Machine Format.
DitsPutDelta	Dits	Put an action delay.

External values used: none

Prior requirements: Should only be called from a Dits application action routine.

Support: Tony Farrell, AAO

A.17 GitPutDelayPar — Set Action delay or timeout from a parameter value.

Function: Set Action delay or timeout from a parameter value.

Description: This routine sets an action delay or timeout from the value of a specified parameter.

Language: C

Call:
(Void) = GitPutDelayPar (parameter, status)

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

- **(>) parameter (char *)** The name of the parameter. The value of this parameter should be able to be interperted as a floating point value. If the value is 0 or negative, no delay will be set.
- **(!) status (StatusType *)** Modified status.

Include files: Git.h

External functions used:

SdpGetd	Sdp	Get a parameter value
GitPutDelay	Git	Set an action delay

External values used: none

Prior requirements: Should only be called from a Dits application action routine in a task which uses the Sdp parameter system.

Support: Tony Farrell, AAO

A.18 GitSimulation — Set a tasks' simulation level and timebase.

Function: Set a tasks' simulation level and timebase.

Description: Examine the SIMULATE_LEVEL parameter. If it is undefined and the variable "envName" is non-zero, then examine the Logical Name (VMS) or Environment Variable (Unix/VxWorks) the name of which is specifed by the "EnvName" variable. If we still don't have a simulation level, then it defaults to "NONE"

If the result of this is that the simulation level is "NONE", then this function returns false.

The "levels" argument determines the acceptable simulation levels. If a value is not acceptable, it is converted to "FULL", which is always acceptable.

The resulting simulation level is put in the SIMULATE_LEVEL parameter and is used to set the "simulation" argument. If the simulation is any other then "NONE" this function will return true.

The timebase is always the value of the TIMEBASE parameter.

Language: C

Call:

(Int) = GitSimulation (exName, levels, simulation, timebase, status)

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

(>) envName (Char *) IF non-zero, then this is the address of the name of a Logical Name (VMS) or Environment

() variable (Unix/VxWorks) which is used to determine the simulation level when the parameter is undefined.

(>) levels (Int) A bit mask indicating the acceptable simulation levels. Any other level is converted to FULL. The following are possible (Taken from GIT_SPEC) -

GIT_M_SIM_BASIC	The level "BASIC" is allowed
GIT_M_SIM_COMMANDS	The level "COMMANDS" is allowed
GIT_M_SIM_STATUS	The level "STATUS" is allowed

"FULL" and "NONE" are always allowed. If zero, then only "FULL" and "NONE" are allowed. Set to GIT_M_SIM_ALLLEVELS to accept all levels

(<) simulation (GitSimulationType *) Indicates the resulting simulation level. One of -

GIT_SIM_NONE	Simulation set to NONE
GIT_SIM_BASIC	Simulation set to BASIC
GIT_SIM_COMMANDS	Simulation set to COMMANDS
GIT_SIM_STATUS	Simulation set to STATUS
GIT_SIM_FULL	Simulation set to FULL

(<) timebase (Float *) The timebase. Only valid if simulation is not equal to `GIT_SIM_NONE`. This is the value in the `TIMEBASE` parameter. A timebase of zero does not make sense, so such a timebase is converted to 1.

(!) status (StatusType *) Modified status.

Function value: 0 if simulation is `GIT_SIM_NONE`, nonzero otherwise.

Include files: Git.h

External functions used:

GitParEnvGetS	Git	The the value of a parameter or environment variable/logical name
strcmp	`CRTL`	Compare two strings
strcpy	`CRTL`	Copy one string to another
ArgGetf	Arg	Get the value of an argument
SdpPutString	Arg	Put the value of a parameter

External values used: none

Prior requirements: Must only be called as part of a Dits application routine. Assumes a parameter system is enabled. (This is case if the task has called GitActivate)

Support: Tony Farrell, `AAO`

A.19 GitTimer — An action routine used to implement an action timer.

Function: An action routine used to implement an action timer.

Description: The first action argument is a count of the number of times to run the timer. Other arguments are used by the user supplied argument creation routines to create arguments to pass to subsidiary action.

This routine first extracts the count. It then calls each of the user supplied argument creation routines (one for each action to be invoked in a sequence. These routines must extract any information they require from the remaining arguments and create an argument structure to be passed to the action they are responsible for.

The action then gets the path to the relevant task and does the entire user requested sequences of actions count times.

To configure this action, the user must supply the address a structure of type GitTimerType as the "code" item when defining a DitsActionMapType which defines this action. This structure has the following user defined elements

TaskName	A character string giving the name of the task to invoke the actions in. This can be any `DRAMA` task including this task.
SeqCount	An integer Count of actions in a sequence
SequenceDetails	Defines details of a sequences. This is an array of Count elements, each of type GitTimerDetailsType, as defined below.
Buffers	An item of type GitPathInfoType which defines the message buffer sizes to be passed to DitsGetPath.
	GitTimerDetailsType has the following user define elements
ActionName	The name of an action to invoke in the target task.
ArgCreateRoutine	The address of a routine of type GitTimerArgRoutineType, which is invoked with the Sds id of the argument structure passed to this action. It should examine this structure and return an argument structure to be passed to the target action.
ClientData	Client data item for the routine.
Arg	Argument to be passed, Set this to 0, it is filled out by the call to the ArgCreateRoutine.

GitTimerArgRoutineType routines have the following definition typedef void (*GitTimerArgRoutineType)(void *ClientData, SdsIdType InArg, SdsId-Type *OutArg, StatusType *status)

Language: C

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

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

(>) status (StatusType *) Modified status

Include files: Git.h

External functions used:

External values used:

Prior requirements:

Support: Tony Farrell, AAO

A.20 GitTimerArgExtract — Basic argument extraction routine for use with GitTimer.

Function: Basic argument extraction routine for use with GitTimer.

Description: This routine can be supplied as an ArgCreate routine to GitTimer. All it does is create an argument structure based on the supplied argument but with the first item extracted and the others renamed Argument1 on wards.

Language: C

Call:
(void) = GitTimerArgExtract (ClientData,InArg,OutArg,status)

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

- **(>) ClientData (void *)** Ignored
- **(>) InArg (SdsIdType)** The argument to the timer actin
- **(<) OutArg (SdsIdType *)** The resulting argument
- **(>) status (StatusType *)** Modified status

Include files: Git.h

External functions used:

External values used:

Prior requirements:

Support: Tony Farrell, AAO

A.21 GitTpiDelete — Delete an Item from the Task/Package Infomation list.

Function: Delete an Item from the Task/Package Infomation list.

Description: Delete an item previously put using GitTpiPut

Language: C

Call:
void = GitTpiDelete (key,item,status)

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

- **(>) key (Int)** The key that was used in the corresponding call to GitTpiPut
- **(!) status (StatusType *)** Modified status. If an item with the specified key cannot be found, status is set to `GIT__TPI_NOTFOUND`;

Include files: Git.h

External functions used:

DitsGetUserData	Dits	Get user data.
DitsPutUserData	Dits	Put user data.
free	CRTL	Release allocated memory.

External values used: none

Prior requirements: GitTpiInit must been called.

Support: Tony Farrell, AAO

A.22 GitTpiGet — Get an Item from the Task/Package Infomation list.

Function: Get an Item from the Task/Package Infomation list.

Description: Return an item previously put using GitTpiPut

Language: C

Call:

void = GitTpiGet (key,item,status)

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

- **(>) key (Int)** The key that was used in the corresponding call to GitTpiPut
- **(<) item (Void **)** The item put by GitTpiPut;
- **(!) status (StatusType *)** Modified status. If an item with the specified key cannot be found, status is set to `GIT__TPI_NOTFOUND`;

Include files: Git.h

External functions used:

DitsGetUserData	Dits	Get user data.

External values used: none

Prior requirements: GitTpiInit must been called.

Support: Tony Farrell, `AAO`

A.23 GitTpiInit — Initialise Task Package Information handling for the current task.

Function: Initialise Task Package Information handling for the current task.

Description: The Task/Package Information routines allow independent packages to use the DitsPutUserData()/DitsGetUserData() routines without conflict to maintian task specific data. All access to DitsPutUserData()/DitsGetUserData() should be via these routines.

The main() function of a task should call this routine to initialise the package. Then, each package which wishes to store common data should call GitTpiPut() to store the address of information it wishes to save. Normally, the address which it wishes to store will be that of a malloced area of memory. The key specified to GitTpiPut() should be unique amoung all packages used by the task (I suggest something based on the message facility used by the package since these numbers should be unique). Later, when the package wishes to retreive such information, it can call GitTpiGet(). GitTpiDelete() can be used to delete an entry.

The use of this package to store task specific common data ensures calling packages will have no problems when ported to systems where all tasks run in the same memory context (such as VxWorks).

Language: C

Call:
void = GitTpiInit (status)

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

(!) status (StatusType *) Modified status.

Include files: Git.h

External functions used:

DitsPutUserData	Dits	Put user data.

External values used: none

Prior requirements: DitsInit has been called.

Support: Tony Farrell, AAO

A.24 GitTpiPut — Put an Item to the Task/Package Infomation list.

Function: Put an Item to the Task/Package Infomation list.

Description: An new entry is created in the Task/Package Information list. They specified key and the user item are stored in the entry.

The key must be a unique integer amoungst all entries for the current DRAMA task.

Language: C

Call:
void = GitTpiPut (key,item,status)

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

- **(>) key (Int)** A key used to lookup this item when calling GitTpiGet. The message facility code for the invoking package is recomended (assuming it only calls this routine once) since it will be unique amoungst all packages.
- **(>) item (Void *)** The item to be stores
- **(!) status (StatusType *)** Modified status. If the item already exists, then status is set to GIT_TPI_EXISTS.

Include files: Git.h

External functions used:

DitsPutUserData	Dits	Put user data.
DitsGetUserData	Dits	Get user data.
GitTpiGet	Git	Get entry item.
malloc	CRTL	Allocate memory.

External values used: none

Prior requirements: GitTpiInit must been called.

Support: Tony Farrell, AAO