ANGLO-AUSTRALIAN OBSERVATORY AAO/DRAMA C++_14
DRAMA Software Report 14
Version 0.6.2

Tony Farrell
19-Nov-1999

DRAMA C++Interface

Contents

Revisions:

24-Feb-2000 DcppMonitor class items which take variable argument lists now support a different format. Add a version of DcppMonitor:Forward which supports a Started Handler. Document use of DitsPutActData() by DcppHandler.

19-Nov-1999 Add DcppShared class and modify methods taking DitsSharedMemInfoType arguments to take this argument. Add DcppHandler:SetTimeout version which allows specification of a timeout handler.

16-Nov-1999 Add new SDS routines to support bulk data (ExportDefined, IsExternal, GetExternInfo, SizeDefined). Document format of DcppBulkTransRoutine routine type. Reformat methods with long argument lists to make things clearer. Add DcppSpawnKickArg and DcppSpawnKickArgUpdate versions of Obey which use these. The use of the DcppHandler increment operator to increase the thread count has been replaced by the NewThread method.

22-Dec-1998 Add bulk data versions of Obey and Kick.

23-Nov-1998 Add tid argument to Obey method.

19-Nov-1998 Document where you can delete a DcppHandler object.

14-May-1998 Add support for intercepting Ers and MsgOut messages. Mention SdsNull.

23-Apr-1997 Add TaskLoggingOn and TaskLoggingOff functions to DcppTask. Add List functions to Arg.

31-Jan-1997 Add LoggingOn, LoggingOff and MGet functions to DcppTask.

25-Sep-1996 DcppHandler object modified to support multiple threads. Sds now has an SdsInsertCell routine which is supported in c++ by an overload of the Insert function. Tidy up Sds usage of const. Sds COut function changed to something which returns the SdsIdType. (Old version still exists in include file). (Have not updated the large example to use the threads mode of DcppHandler. This should be done at some stage).

29-Jul-1996 DcppBuffers may not be initialised from and return structures of type DitsPathInfoType. DcppTask function SetFLowControl added.

06-May-1996 Add Sdp routine equivalent of SdpPutStruct. Ensure Sdp and Arg character string name arguements are const.

21-Jun-1995 Added Git interfaces. Added Forget versions of DcppTask and Dcpp Monitor functions. General update and tidy up of document.

14-Aug-1995 Added LosePath method to DcppTask. Added DcppUfaceCtxEnable routine. Added GitInt and GitReal classes.

18-Sep-1995 Added Sds interface.

18-Oct-1995 Reformat as a LaTeXdocument.

1 Introduction

This document describes an experimental C++ interface to DRAMA.

Note that all normal DRAMA C routines are usable in C++ in the normal way. This interface is additional to that provided by the DRAMA C interface.

A C++ interface is largely a set of C++ Classes (or data types) and the operations (methods) which may be applied on them.

The "dcpp" items are in "DUL" library. You will need to link against the DUL library and put "DUL_DIR" in your include file search path. The other items are in the libraies they naturally belong in (Class **SdsId** in Sds for example).

2 Sds and Arg

The class **SdsId** provides a C++ interface to Sds ([3]). Each C++ **SdsId** type variable represent an Sds Id (C type `SdsIdType`), not a complete Sds structure to which many Sds Id's may point. Each operation in the Sds C library which allocates an Sds Id maps to a constructor in the **SdsId** class (this mapping is not one-to-one). An additional constructor is provided to import an existing `SdsIdType` into an **SdsId** variable. Member functions are provided for all the standard Sds operations.

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

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

```
            SdsFreeId(fieldId,&ignore);
        }
        /* Tidy up from read */
        ignore = STATUS__OK;

        SdsReadFree(fileId,&ignore);
        SdsFreeId(fileId,&ignore);
    }
}
```

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

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

The **SdsRead(3)** call occurs within the first constructor, while the **SdsFind(3)** operation occurs in the second. In both cases, flags kept in the **SdsId** variable indicate how the item was allocated, allowing the destructors invoked implicitly at the end of the routine to clean up. When routines which must navigate large structures are considered, the benefits are even more dramatic.

2.1 Assignment and copying

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

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

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

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

In the case of the **ShallowCopy**, in which the actual Sds Id is copied, you must indicate if the copy variable is to outlive the source. This is necessary since if it is to do so then the copy must be responsible for any required deletion and freeing operations, not the original variable. The

function will make the necessary modifications to both variables to ensure the destructors work as required.

The **DeepCopy** function uses **SdsCopy(3)** to create a new structure which is a copy of the source structure.

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

2.2 Importing from SdsIdType variables

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

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

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

You must use the three flags to tell **SdsId** how to handle this id in it's destructor. If `free` is set true, the id is free-ed using **SdsFreeId(3)**. If `del` is true, then the item will be deleted using **SdsDelete(3)** or **SdsReadFree(3)**. The `readfree` flag is set true to indicate that **SdsReadFree(3)** should be used instead of **SdsDelete(3)**. All these flags have default values of false, telling the destructor to do nothing.

The following bit of code uses a call to **GitArgGetStruct(3)** to return an Sds id. We then setup an **SdsId** item to access it. Since the id returned by **GitArgGetStruct(3)** must be free-ed, but not deleted by the user when he is finished with it, we set the `free` flag `true` and leave the rest of the flags at their default `false` values.

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

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

2.3 Exporting to SdsIdType variables

In order to export the Sds id represented by an **SdsId** variable to an `SdsIdType` variable, use the member function **COut**.

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

This function returns the appropriate `SdsIdType` in the `id` variable. You must indicate if the `SdsIdType` variable will outlive the **SdsId** variable by setting the `outlives` flag true. This will ensure that the destructor for the **SdsId** variable does not free the id. You can make use of the other arguments to determine what you should do with the id in the case of it outliving the source. The flag pointers are optional and if set to the default of 0, are ignored, but otherwise indicate what the destructor would have done.

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

2.4 Null Sds items

In many cases the DCPP and other functions which follow will ask for a reference to a **const SdsId**. You can if desirable, specify here the address of a Null Sds item, named **SdsNull**. This item is a variable of type **SdsId** which contains the null Sds Id (0). Use this where you don't want to supply an actual Sds id. For example, it can be used for the **arg** argument to **DcppTask::Obey()**.

2.5 Static and Global variables

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

```
static SdsId paramId;      /* Static item using default constructor */
...
/* Create the item an copy it to the static item*/
SdsId top("TOP",SDS_STRUCT,status);
paramId.ShallowCopy(top,true); /* paramId outlives top */
```

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

2.6 Arg

The **Arg** class is derived from **SdsId**. Only two constructors are available. One of these is identical to the **SdsId** import constructor described in 2.2 (This also implements the default constructor in both cases, using argument defaults).

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

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

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

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

3 Sdp

The **Sdp** type is a simple interface to the System **Dits** Parameter system ([2]). There are no constructors, just static member functions which make use of overloading to select the appropriate interface to Sdp.

You should use calls like these-

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

4 Git

A number of classes are defined in the Git include file. The base type **Git** provides a simpler way of specifying the various flags used by the equivalent of the **GitArgGet*** functions ([4]).

4.1 GitBool

This class implements a boolean type which includes a member function to fetch values from an Sds structure using **GitArgGetL(3)**. This class provides a simpler interface to fetching boolean values from string or integer values in Sds structures.

The following code shows the declaration of a **GitBool** type and the fetching of the value from an Sds item. (Note that since a conversion exists for `SdsIdType` to **SdsId** we can use **DitsGetArgument(3)** directly to get the Sds id.)

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

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

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

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

```
class ParkBool : public GitBool {
    private:
        static const GitLogStrType lookupTable[];
        const GitLogStrType * Lookup() { return lookupTable;};
    public:
        ParkBool(const SdsId & Id, const char * const Name,
                 const int Position, StatusType * const status,
                 const int Flags =  Git::Upper|Git::Abbrev))
                //Use Get not constructor, see C++ Ref manual r.12.7
  : Get(Id,Name,Position,status,false,Flags){ }
};
/* Define the static item defined in ParkPool */
const GitLogStrType ParkBool::lookupTable[] = {
    { "PFA", "ZENITH"},
    { "TRUE", "FALSE"},
    { 0, 0 } };
...
/* Using the value*/
ParkBool Posit(DitsGetArgument(), "ZENITH",1,status);
if (Posit)
...
```

4.2 GitEnum

This class implements an enumerated type which includes a member function to fetch values from an Sds structure using **GitArgGetS(3)**. This class provides a simpler interface to fetching enumerate values from strings in Sds structures. This is a virtual class, the user must

provide a class which inherits this class and provides implements of the functions **SetValue** and **Lookup()**.

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

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

```
class RecType : public GitEnum {
    private:
      /* enum possibilities */
      enum recEnum { Record=0, Dummy, Glance, Invalid };
      /* value contains the actual value */
      recEnum value;
      /* Lookup table returns the list of strings */
      static const char * const lookupTable[];
      /* Conversion operator, given enum, return an int */
      operator int() const { return ((int))value));

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

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

```
          const int Flags=Git::Upper|Git::Abbrev) {

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

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

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

4.3 GitInt and GitReal

These classes implement integer and floating point types respectively which include member functions to fetch values from an Sds structure using **GitArgGetI(3)/GitArgGetD(3)**. These classes provide simpler interfaces to fetching integer and real values from Sds structures. (Note that at present, the implementation of these types is probably not complete, not all integer/real operations can be performed, although you can convert them to int/double as appropriate.

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

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

```
/*
 * Repeat mode count integer
 */
class RepCount : public GitInt {
    private:
      virtual const long int * Range() { return range; }
      static const long int range[];
    public:
      /* Constructor with automatic get */
      RepCount(
            const SdsId& Id,
            const int Position,
            StatusType * const status,
```

```
            const int  Default = 1,
            const int Flags = Git::KeepErr) {
              GitInt::Get(Id,"REPEAT_COUNT",Position,
                    status,Default,Flags);
      }
      /* Simple constructor */
      GctRepCount(const long int def = 1) : GitInt(def){}
      /* Get operator */
      void Get(
            const SdsId& Id,
            const int Position,
            StatusType * const status,
            const int Default = 1,
            const int Flags = Git::KeepErr) {
              GitInt::Get(Id,"REPEAT_COUNT",Position,
                    status,Default,Flags);
      }
      /* Pre-decrement operator */
      GctRepCount operator--() {
            long int value = *this;
            *this = GctRepCount(value-1);
            return value;
      }
};
/*
 * Define static item defined above
 */
const long int GctRepCount::range[] = { 1,100000};

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

5 Dcpp

The **Dcpp** set of classes implement a C++ interface to the **Dits** [2] task communication functions. The scheme implemented allows you to send a message specifying callback functions to be invoked when responses are received.

Assuming the thread of control is setup correctly, your callback functions will be invoked automatically. They return a code which indicate whether of not they are expecting further messages on the current thread of control.

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

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

The other type of control thread is **UFACE** context routines. Only people writing user interface code need be concerned with these threads. See [1] and [2] for more details on **UFACE** context. To use **DcppTask** based communications from **UFACE** threads, you should use the **DcppUfaceCtxEnable()** routine in place of the **DitsUfaceCtxEnable(3)**.

5.1 A task communication example

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

```
static DcppHandlerRet StartObey(                                    ◇1
                DcppVoidPnt ClientData,
                StatusType * const status);

static DcppTask ticker("TICKER",0,"DITS_LIB:ticker");               ◇2
static DcppHandler Handler;                                         ◇3

static void DpRun(StatusType * const status)                        ◇4
{
    Handler.Install(status);                                        ◇5
    if (ticker.GetPath(status,StartObey) ==
                              DcppReschedule)
    {
        DitsPutRequest(DITS_REQ_MESSAGE,status);
    }
}

static DcppHandlerRet StartObey(                                    ◇6
                       DcppVoidPnt ClientData,
                       StatusType * const status)
{
    return(ticker.Obey("TICK",status));
}
```

At ◇**1** is the prototype for the **StartObey()** function. This function is defined later and used as the **SuccessHandler** for the **GetPath** operation. At ◇**2** we see the definition of a **DcppTask** object named `ticker`. This is specified with a name of "TICKER" - the name the task is expected to have registered under if it is already running. A location of 0, (the null pointer) indicates we expect the task to be on the local machine and will load in on the local machine if it is loaded. The third argument specifies the file from which the task is loaded if it is not already running.

At ◇**3** we define a **DcppHandler** object. We don't specify any callback routines, which means that any errors or completion subsidiary actions will cause the action to complete.

◇**4** is the definition of the **DpRun** function. This function is an Obey Handler, which would be specified in a **DitsActionMapType** variable passed to **DitsPutActionHandlers(3)**. An obey message will result in this function being invoked. At ◇**5** we install the handler, which will result in **DcppHandler** handling future reschedules of this action. We are a bit lazy in that we don't **DeInstall** this object at any stage. This is OK as a **DeInstall** is only required if we are to continue the action under our own control. On the next line, we initiate a **GetPath** operation

on the `ticker` task object. We specify **StartObey()** as the success handler. By not specifying an **ErrorHandler**, the action will complete on error. Now **GetPath** will always result in a message and return `DcppReschedule` unless an error occurs, so we immediately request a reschedule. **GetPath** automatically tries to load the task using the information available to it if it cannot find the task.

◇**6** is the definition of the **StartObey()** handler. This will be invoked when the **GetPath** operation completes successfully. In it, we simply send the Obey message. Note that this routine is invoked in the context of the action by a routine in the **DcppHandler** class. **StartObey()** should return `DcppReschedule` if anything it does requires the action to be rescheduled, which is also what **Obey** will return if it works, so we just return the value of **Obey**. No success, trigger or error handlers are specified which will result in the action completing when the Obey's completion message arrives. The previous **DcppHandler** object remains installed until the action exits.

By default, a **DcppHandler** will keep track of one thread of messages. For example, the **GetPath** operation followed by the **Obey** message. If you wish to have multiple threads active at one time, for example, if you start a sequence of **GetPath** operations, each followed by an **Obey**, you should indicate to the **DcppHandler** object that there are multiple threads. For each thread after the first, you should invoke the **DcppHandler NewThread** method. [1]. The **DcppHandler** object will then continue to reschedule the action until all threads have returned DcppFinished.

5.2 A simple example of parameter monitoring.

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

[1]Previously, the increment operator (either pre or post increment) was used for this. It is currently still available but will be removed at some stage

```
static void DpMon(const char * const name,                          ◇1
                  const SdsCodeType type,
                  const DcppVoidPnt value,
                  const DcppVoidPnt ClientData,
                  StatusType * const status)
{
    MsgOut(status,"Parameter %s changed",name);
}

DcppMonitor Monitor(&ticker);                                       ◇2
DcppHandler Handler2;                                               ◇3

static void DpTest (StatusType *status)                             ◇4
{
    Handler2.Install(status);

    Monitor.Monitor(DpMon,0,0, false,1,status,"PARAM1");            ◇5
    DitsPutRequest(DITS_REQ_MESSAGE,status);
}

static void DpTestKick(StatusType *status)                          ◇6
{
    Monitor.Cancel(status,DcppTask::DiscardResponse);
}
```

At ◇**1** we have the definition of the routine that will be invoked automatically when the parameter value has changed. In this example, it only outputs a simple message. At ◇**2** we have the definition of the monitor object. We have specified the address of a **DcppTask** object named `ticker` that would have been declared previously. At ◇**3** we define a **DcppHandler** object.

◇**4** is the definition of the action handler routine that will be invoked to do the work. After installing the handler, we start the monitor at ◇**5** specifying the routine `DpMon()` to be invoked when the parameter changes and `PARAM1` as the name of the parameter to monitor. We assume at this point that some where a **GetPath** operation has already been performed on the `DcppTask` object named ticker. That is all there is to it. The rest of the work is done within the **DcppMonitor** and **DcppHandler** classes.

The one thing left to do is to cancel monitoring if necessary. In this example, a kick handler is provided at ◇**6**. All we need to is send a cancel to the monitor. By specifying the **DiscardResponse()** routine as the success handler, we are saying we want to ignore any response, hence we have no need to reschedule.

5.3 A more complex example.

A more complex example is attached. This is a re-implementation of the `CTEST` program described in [1]. The size of the program source has been reduced by about half.

6 Efficiency Considerations.

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

6.1 SdsId

The **SdpId** class adds one item (byte or word size, depending on the compiler and target) to each Sds id. This is used to maintain the flags which indicate if the item should be delete, frdeed etc.

An additional one word is for the virtual function lookup pointer.

Constructors must add code to set these items up, but otherwise, there is generally no overhead added to the run time efficiency of the underlying Sds calls.

6.2 Arg

The **Arg** class adds no overhead over the **SdsId** overhead for each Sds id.

6.3 Sdp

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

6.4 Git

The **GitBool**, **GitEnum**, **GitInt** and **GitReal** classes add at least one pointer to each data item - used to find the virtual function lookup table. Other then this, they are probably not any more inefficient then using the equivalent **GitArg*(3)** functions directly. In the case of **GitEnum**, most of the work is in the declarations, not in the resulting code. I believe the example code given (**RecType**) would use no more runtime code then a direct call to the **GitArgGetS(3)** function and the associated checks. You do through get much more compile time checking.

6.5 DcppTask and DcppHandler

A well-defined amount of overhead is added to the initiation of each message being the allocation and filling in of a structure to maintain details associated with a transaction. This structure is dynamically allocated. If this proves a problem, then the default allocator could be replaced with a more appropriate algorithm.

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

7 The Classes

7.1 SdsId — A class that provides a C++ interface to the C Sds library.

Derivation: This is a base type.

Include File: sds.h

Description: The major benefit of a C++ interface to the Sds library is the automatic deleting of Sds items and freeing of Sds id's when the variable accessing the item goes out of scope. In addition, access to many Sds operations is simpler with this class.

This class contains a large number of constructors generally corresponding to each case in the original Sds C library where a new Sds id is generated. Operations such as **SdsNew(3)**, **SdsIndex(3)**, etc. all have corresponding constructors in this class. Each item of this type corresponds to a single Sds id, not necessarily a single Sds item or structure, to which there may be multiple Sds id's pointing.

Every time a new item of this type is constructed, consideration is made of whether the item should be deleted and the id free-ed when the variable goes out of scope. Generally, any id allocated is free-ed, and any internal top level item deleted when the associated variable goes out of scope. In the case of items created using **SdsRead(3)**, **SdsReadFree(3)** is called before the id is free-ed. As a result, you only have to consider deleting structures and freeing id's in a couple of special cases. (when importing or exporting an id to/from a **SdsIdType** and when doing a shallow copy (see below)).

Assignment and copying of variables of this type has been prohibited by making the corresponding operators private to the class. This is done because a) it is not always clear what the user may want done (a deep or shallow copy for example) and b) handling of errors requires a status variable which is not available in these operations. One result of this is that you cannot pass variables of this type to subroutines by value, you must pass them by either pointer or reference.

The member functions **ShallowCopy** and **DeepCopy** provide explicit control of copy and assignment operations. In the former case, you must indicate if the copy variable is to outlive the source. This is necessary since if it is to do so then the copy must be responsible for any required deletion and freeing operations, not the original variable. The function will make the necessary modifications to both variables to ensure the destructors work as required. You can also make use of the constructor which constructs a deep copy of an existing item.

Constructors:

- **SdsId(SdsIdType id = 0, bool free=false, bool del = false, bool readfree = false);**
  A Constructor that builds an SdsId variable based on an an existing Sds id. Free indicates we should free id when we destroy the variable, del that we should delete the Sds item before freeing the id and readfree that the item was allocated by **SdsRead(3)**. Note that the id = 0 case (equivalent to a default constructor) is normally only used when we want an argument to pass to a routine which wants to return an SdsId value

or when using static of global variables of this type. In these cases, the **ShallowCopy** or **DeepCopy** methods would be used to set the id.

- **SdsId(void * data, StatusType * status, bool import = false);**
  A Constructor that builds an SdsId variable by doing an access or import operation to get the id (SdsImport(3)/SdsAccess(3)).
  We use the one constructor since both operations take the same arguments. Set the "import" flag true to do an Import operation instead of an Access operation.
- **SdsId(const void * data, StatusType * status);**
  A Constructor that builds an SdsId variable by doing an import operation to get the id. This alternative constructor to the previous one is invoked when the original data is const. You cannot do an access operation on a const item (SdsImport(3)).
- **SdsId(const char * filename, StatusType * status);**
  A constructor that builds an SdsId variable by reading a structure from a file (SdsRead(3)).
- **SdsId(const SdsId &parent, const char * name, SdsCodeType code, StatusType * status, long nextra = 0, char * extra = 0);**
  A constructor that creates a new (non-array) child item of the specified "parent" item (SdsNew(3)).
- **SdsId(const char * name, SdsCodeType code, StatusType * status, long nextra = 0, const char * extra = 0);**
  A constructor that creates a new (non-array) top level item (SdsNew(3)).
- **SdsId(const SdsId &parent, const char * name, SdsCodeType code, long ndims, const unsigned long *dims, StatusType * status, long nextra = 0, const char * extra = 0);**
  A constructor that creates a new array child item of the specified "parent" item (SdsNew(3)).
- **SdsId(const char * name, SdsCodeType code, long ndims, unsigned long *dims, StatusType * status, long nextra = 0, const char * extra = 0);**
  A constructor that creates a new array top level item (SdsNew(3)).
- **SdsId(const SdsId & array_id, long nindicies, const unsigned long * indicies, StatusType * status);**
  A constructor that create a variable pointing to a cell of an existing array id (SdsCell(3)).
- **SdsId(const SdsId & source, StatusType * status)**
  A constructor which creates a variable pointing to a copy of the specified variable. This constructor can be considered a "copy-constructor" but it is NOT the standard copy-constructor (SdsCopy(3)).
- **SdsId(const SdsId & source, const char * name, StatusType * status);**
  A constructor which creates a variable pointing to a named item of an existing structured Sds item (SdsFind(3)).
- **SdsId(const SdsId & source, long index, StatusType * status);**
  A constructor which returns an id to an existing structured item indexed by position (SdsIndex(3)).

Operators: The assignment operator and copy constructor are private, preventing assignment and copying of items of this type, since these operations are not sensible for this type. No other operators are provided. See comments in the "Description" section above and the **ShallowCopy** and **DeepCopy** member functions.

Conversion Operators:

- **operator SdsIdType() const;**
  Returns the Sds Id associated with the variable. Note, if the operator is used, it is expected that the original variable outlives the usage of the return value.
- **operator bool(void) const;**
  Returns true if the underlying Sds id points to a valid (non-zero) Sds id.

Overloadable Methods:

- **void Delete(StatusType * status);**
  This method is used to Delete an Sds item and free the id as if the destructor for the SdsId item has been run. This can be used where you know the item is no longer required but the destructor will not be run for a while.
- **void Get(unsigned long length, void * data, StatusType * status, unsigned long *actlen = 0, unsigned long offset=0)**
  Get the contents of an Sds item into the buffer "data", which is of length "length".
- **void Put(unsigned long length, const void * data, StatusType * status, unsigned long offset=0);**
  Put the contents of an Sds item from the buffer "data", which is of length "length".

Methods 1: These methods are equivalent to various Sds C library functions.

- **void Code(SdsIdType * code, StatusType * status);**
  Return the Sds type code of the item.
- **void Dims(long * ndims, unsigned long * dims, StatusType * status)**
  Return the number of dimensions and the dimensions of the Sds item.
- **void Export(unsigned long length, void * data, StatusType * status);**
  Export the Sds item into the buffer "data" of length "length".
- **void ExportDefined(unsigned long length, void * data, StatusType * status);**
  Export the Sds item into the buffer "data" of length "length", defining any SDS items which have not been defined.
- **void Extract(StatusType * status);**
  Extract a child Sds structure from it's parent structure.
- **void Flush(StatusType * status)**
  Indicate to Sds that an item being access by pointer has been updated.

- **void GetExternInfo(void **data, StatusType * status) const;**
  If the SDS item is an external item, the address of the buffer handling the item is returned.
- **void GetExtra(unsigned long length, char * extra, StatusType * status, unsigned long *actlen = 0);**
  Get any extra data associated with the item into the buffer "extra" of length "length".
- **void GetName(char * name, StatusType * status);**
  Return the name of the sds item.
- **void Info(char * name, SdsCodeType * code, long * ndims, unsigned long * dims, StatusType * status)**
  Return details about the Sds item, being the "name", the type "code", the number of dimensions "ndims" and the actual dimensions "dims".
- **void Insert(SdsId & to_insert, StatusType * status);**
  Insert an existing Sds item into this item.
- **void Insert(SdsId & to_insert, long * ndims, const unsigned long *ndims StatusType * status);**
  Insert an existing Sds item into a specified cell of a structured array.
- **void IsExternal(int *external, StatusType * status) const;**
  The variable `extern` is set to indicate if the SDS item is an external item.
- **void List(StatusType * status) const ;**
  List the contents of the Sds item on the standard output device.
- **void Pointer(void **data, StatusType * status, unsigned long * length = 0)**
  Return a pointer to the data area of an Sds item. Note if you update the item, you should use **Flush** to notify Sds you have done so.
- **void PutExtra(long nextra, const char * extra, StatusType * status);**
  Put the extra information associated with an item.
- **void Rename(const char * name, StatusType * status);**
  Rename the Sds item to the specified name.
- **void Resize(long ndims, const unsigned long *dims, StatusType * status)**
  Resize the sds item as specified.
- **void Size(unsigned long * bytes, StatusType * status) const ;**
  Return the number of bytes required to export the Sds item.
- **void SizeDefined(unsigned long * bytes, StatusType * status) const ;**
  Return the number of bytes required to export the Sds item, assuming any currently undefined have been defined. This is the size of the buffer required by **ExportDefined**.
- **void Write(const char * filename, StatusType * status) const;**
  Write the contents of the Sds item to the specified file.

Methods 2: Miscellaneous methods

- **SdsIdType COut(bool outlives, bool * free = 0, bool * del= 0, bool * readfree = 0);**
  Setup for and obtain details for the return of this item using an **SdsIdType**. If outlives is true, the **SdsIdType** item will outlive this item and we leave the freeing of the id etc. up to the caller, who can use the other variables to work out what to do. If outlives is false, this method can be used as an enquiry;
- **void Outlive()** Forces the actual Sds id to outlive the variable. This item should be used with care as it may result in id's not being tided up, but is useful in some situations.
- **void ShallowCopy (SdsId & source, bool outlives);**
  Shallow copy from a "source" of type SdsId. If the new variable will outlive "source", we should set the "outlives" flag true, otherwise set it false.
  By a shallow copy, we mean we use the same Sds id referenced by "source".
  Any existing item pointed to by this variable is destroyed.
- **void ShallowCopy (SdsIdType source, bool free=false, bool del = false, readfree = false);**
  Shallow copy from a "source" of type SdsIdType. We must set "free" true if we want the Sds id to be free-ed when this variable goes out of scope. Similarly, set "del" true if we want the item deleted when the variable goes out of scope. Set "readfree" true if the item was created using **SdsRead(3)**.
  Any existing item pointed to by this variable is destroyed.
- **void DeepCopy (const SdsId &source, StatusType * status)**
  Create a deep copy of "source". A deep copy uses **SdsCopy(3)** to create a new item.
  Any existing item pointed to by this variable is destroyed.
- **void DeepCopy (SdsIdType source, StatusType * status)**
  Create a deep copy of "source". A deep copy uses **SdsCopy(3)** to create a new item.
  Any existing item pointed to by this variable is destroyed.

7.2 Arg — A class that provides a C++ interface to the C Arg library.

Derivation: Arg←SdsId

Include File: arg.h

Description: The major benefit of a C++ interface to the Arg library is the automatic selection of appropriate routines for each argument type. It is based on the **SdsId** type.

Constructors:

- **Arg(SdsIdType id = 0, bool free=false, bool del = false, readfree = false);**
  A Constructor that builds an Arg variable based on an an existing Sds id. Free indicates we should free id when we destroy the variable, del that we should delete the Sds item before freeing the id and readfree that the item was allocated by **SdsRead(3)**.
- **Arg (bool New, StatusType * status, char * const name = "ArgStructure");**
  Create a new arg item. The New argument can have any value, it is just used to make this constructor different from the above one when the trailing arguments are defaults. The name of the structure can be change if required.

Operators: The assignment operator and copy constructor are private, preventing assignment and copying of items of this type, since these operations are not sensible for this type. No other operators are provided. See comments in the "Description" section of the class **SdsId** and the **ShallowCopy** and **DeepCopy** member functions of that class.

Methods:

- **void New (StatusType * status, const char * name = "ArgStructure");**
  Create a new item an existing Arg variable.
  Any existing item pointed to by this variable is destroyed.
- **void ToString(int maxlen, int *length, char *string, StatusType * status);**
  Write a string representation of the structure into "string", which is of length "maxlen". The actual length of the return string is written into "length".
- **void Put (const char *name, ScalerType value, StatusType * status);**
  Put the value of the item "name". The type of "value" can be one of the standard Sds scaler values.
- **void Put (const char *name, const char * value, StatusType * status);**
  Put the value of the item "name" from a character string.
- **void Get (const char *name, ScalerType * value, StatusType * status);**
  Get the "value" of the item of the given "name". The type of "value" can be one of the standard Sds scaler values.
- **void Get (const char *name, long len, char * value, StatusType * status);**
  Get the "value" of the item of the given "name" as a character string.
- **void List(StatusType * status) const ;**
  List the contents of the Sds item on the standard output device.

- **static void List(const SdsId &id, unsigned buflen char *buffer, ArgListFuncType func, void *client_data, StatusType * status) const ;**
  List the contents of the Sds item using the function "func" as the output function. The is really just an interface to "ArgSdsList(3)" for any SdsId or Arg type item. See "ArgSdsList(3)" for full details.

7.3 Sdp — A class that provides a C++ interface to the C Sdp library.

Derivation: This is a base type.

IncludeFile: Sdp.h

Description: The major benefit of a C++ interface to the Sdp library is the automatic selection of appropriate routines for each argument type.

There is no actual object and hence no constructor. Just static member functions which provide access to the C interface using function overloading.

To use these functions use calls like

```
Sdp::Put("FRED",1,status);
```

Constuctors: No constructors are provided. See the description.

Static Members:

- **void CreateItem (const SdsId & parsys, SdsId & item, StatusType *status);**
  Create a parameter in the parameter system specified by parsys (normally (SdsIdType)DitsGetParId()). The specified item is inserted in the parameter system.
- **void Put (const char *name, ScalerType value, StatusType * status);**
  Put the value of the item "name". The type of "value" can be one of the standard Sds scaler values.
- **void Put (const char *name, const char * value, StatusType * status);**
  Put the value of the item "name" from a character string.
- **void Put (const char *name, const SdsId & value, StatusType * status);**
  Put the value of the item "name" from an Sds id.
- **void Put (const char *name, bool copy, StatusType * status, bool create = true);**
  Insert the specified Sds structure into the parameter system as the value of the item "name". If copy is true, then we create copy the item and insert the copy, otherwise we insert the supplied item. If create is true, then it is acceptable to create a new parameter of the given name.
- **void Get (const char *name, ScalerType * value, StatusType * status);**
  Get the "value" of the item of the given "name". The type of "value" can be one of the standard Sds scaler values.
- **void Get (const char *name, long len, char * value, StatusType * status);**
  Get the "value" of the item of the given "name" as a character string.
- **void Get (const char *name, SdsId & value, StatusType * status);**
  Get the "value" of the item of the given "name" as an Sds item.

7.4 Git

A number of classes have been created to provide a C++ interface to the C Git library.

7.4.1 Git — A class defining the flags used in the other Git C++ interfaces.

Derivation: This is a base type.

Include File: Git.h

Description: This class simply redefines the various Git flags, taking advantage of C++ features to simplify their use.

The following flags are defined.

Upper Equivalent to `GIT_M_ARG_UPPER`

Lower Equivalent to `GIT_M_ARG_LOWER`

KeepErr Equivalent to `GIT_M_ARG_KEEPERR`

Abbrev Equivalent to `GIT_M_ARG_ABBREV`

LastBit Equivalent to `GIT_M_ARG_LASTBIT`

Normal usage will see individual flags specified like this

```
Git::Upper
Git::Abbrev
```

while flags can be or-ed like this

```
Git::Upper|Git::Abbrev
```

No constructors or methods are defined.

7.4.2 GitBool — A class providing a C++ wraparound of GitArgGetL(3).

Derivation: GitBool←Git.

Include File: Git.h

Description: This class implements a boolean type which includes the operation Get, to fetch its value from an Sds structure. By default, the string combinations "`YES/NO`" and "`TRUE/FALSE`" are accepted in the Sds item as boolean values, any input string being converted to upper case and abbreviations being accepted. The Sds item could also represent logical values using integer values. There is also a constructor which does the Get operation.

By overriding the virtual function Lookup in a inheriting class, you can supply alternative lists of True/False strings.

Constructors:

- **GitBool();**
  Constructs a GitBool type with a value of false.
- **GitBool (const SdsId& Id, const char * Name, int Position, StatusType * status, bool Default = false, int Flags = (Git::Upper|Git::Abbrev));**
  Constructs a GitBool type and sets it's value by doing a **GitArgGetL(3)** operation using the Sds item supplied. "Name" is the name of the item in the structure and "Position" is the position of the item if the name is not supplied. See **GitArgGetL(3)** for more details.

Conversion Operators:

- **operator bool() const;**
  Return the value of the GitBool type as a bool.
- **operator int() const;**
  Return the value of the GitBool type as an int.

Overloadable Methods:

- **const GitLogStrType * Lookup()**
  Returns the address of a variable of type `GitLogStrType`, used as the lookup table for True and False value string equivalents. See **GitArgGetL(3)** for details. Note that the default implementation of this method is private to GitBool, but this does not stop you overriding it.

Methods:

- **void Get(const SdsId& Id, const char * Name, int Position, StatusType * const status, bool Default = false, int Flags = (Git::Upper|Git::Abbrev));**
  Set the value by doing a **GitArgGetL(3)** operation using the Sds item supplied. "Name" is the name of the item in the structure and "Position" is the position of the item if the name is not supplied. See **GitArgGetL(3)** for more details.

7.4.3 GitEnum — A class providing a C++ wraparound of GitArgGetS for getting Enum values.

Derivation: GitEnum←Git.

Include File: Git.h

Description: This class implements a facility for fetching the value of an enumerated type from an Sds structure using **GitArgGetS(3)**. This is a virtual class - the user must provide a class which inherits this class and provides implementations of the functions **SetValue()** and **Lookup()**.

Constructors:

- **GitEnum();**
  The default constructor. You can't invoke this item directly since this class is virtual. Since there are no data members associated with this item, there is nothing to be setup.

Overloadable Methods:

- **void SetValue(const unsigned int);**
  Takes an unsigned integer and sets the object to the equivalent enum value. It is up to the function to decide what to do with out of range items, but it is suggest that the enum should have an invalid item.
  Out of range items could cause the object to be set to the invalid item
- **const char * const * Lookup();**
  Returns the address of a variable of an array of strings terminated with a null value, used as the lookup table for the string equivalents of acceptable enumerated values. This to **GitArgGetS(3)** as it's `values` argument. See **GitArgGetS(3)** for more details.

Methods:

- **void Get(const SdsId& Id, const char * Name, int Position, StatusType * const status, const char *Default = false, int Flags = (Git::Upper|Git::-Abbrev));**
  Get the value by doing a **GitArgGetS(3)** operation using the Sds item supplied. "Name" is the name of the item in the structure and "Position" is the position of the item if the name is not supplied. The `values` argument to **GitArgGetS(3)** is provided by invoking the `Lookup()` function provided by the derived class while the value is set by invoking the `SetValue()` function provided by the derived class. See **GitArgGetS(3)** for more details.

7.4.4 GitInt — A class providing a C++ wraparound of GitArgGetI(3)

Derivation: GitInt←Git.

Include File: Git.h

Description: This class implements a facility for fetching the value of an integer type from an Sds structure using **GitArgGetI(3)**.

By default, there is no range limitation on the item but when inheriting this class, a derived class can add a range restriction by proving an implementation of the **Range** method.

Constructors:

- **GitInt(def = 0);**
  Constructs a GitInt type with a value specified by "def".
- **GitInt (const SdsId& Id, const char * Name, int Position, StatusType * status, long int Default = 0, int Flags = 0);**
  Constructs a GitInt type and sets it's value by doing a **GitArgGetI(3)** operation using the Sds item supplied. "Name" is the name of the item in the structure and "Position" is the position of the item if the name is not supplied. See **GitArgGetI(3)** for more details.

Conversion Operators:

- **operator long int() const;**
  Return the value of the GitInt variable as a long integer.

Overloadable Methods:

- **const long int * Range()**
  Range() returns the address of an array of long integers with two entries, giving the minimum and maximum values for the value being fetched.

Methods:

- **void Get(const SdsId & Id, const char * Name, int Position, StatusType * status, long int Default = 0, int Flags = 0)**
  Fetch the value of a GitInt type and sets it's by doing a **GitArgGetI(3)** operation using the Sds item supplied. "Name" is the name of the item in the structure and "Position" is the position of the item if the name is not supplied. The value returned by `Range()` will be supplied as the `range` argument to **GitArgGetI(3)**. See **GitArgGetI(3)** for more details.

7.4.5 GitReal — A class providing a C++ wraparound of GitArgGetD(3)

Derivation: GitReal←Git.

Include File: Git.h

Description: This class implements a facility for fetching the value of an real type from an Sds structure using **GitArgGetD(3)**.

By default, there is no range limitation on the item but when inheriting this class, a derived class can add a range restriction by proving an implementation of the **Range** method.

Constructors:

- **GitReal(def = 0.0);**
  Constructs a GitReal type with a value specified by "def".

- **GitReal (const SdsId& Id, const char * Name, int Position, StatusType * status, double Default = 0.0, int Flags = 0);**
  Constructs a GitReal type and sets it's value by doing a **GitArgGetD(3)** operation using the Sds item supplied. "Name" is the name of the item in the structure and "Position" is the position of the item if the name is not supplied. See **GitArgGetD(3)** for more details.

Conversion Operators:

- **operator double() const;**
  Return the value of the GitReal variable as a double.
- **operator float() const;**
  Return the value of the GitReal variable as a float.

Overloadable Methods:

- **const long int * Range()**
  Range() returns the address of an array of doubles with two entries, giving the minimum and maximum values for the value being fetched.

Methods:

- **void Get(const SdsId & Id, const char * Name, int Position, StatusType * status, double Default = 0.0, int Flags = 0)**
  Fetch the value of a GitReal type and sets it's by doing a **GitArgGetD(3)** operation using the Sds item supplied. "Name" is the name of the item in the structure and "Position" is the position of the item if the name is not supplied. The value returned by `Range()` will be supplied as the `range` argument to **GitArgGetI(3)**. See **GitArgGetD(3)** for more details.

7.5 DcppBuffers — A type to contain message the buffer sizes as required for use in calls to DitsGetPath(3).

Derivation: This is a base type.

Include File: dcpp.h

Description: This class provides a convenient way to pass buffer sizes around.

Constuctors:

- **DcppBuffers(long int MessageBytes = 800, long int MaxMessages = 2, long int ReplyBytes = 800, long int MaxReplies = 2);**
  This constructor creates a DcppBuffers variable. Buffer sizes default to values appropriate for simple tasks.
- **DcppBuffers(const DitsPathInfoType & info);**
  Initialise a DcppBuffers variables from a DitsPathInfoType variable.

Conversion Operators:

- **operator const * DitsPathInfoType() const;**
  Returns the address a structure for passing to DitsPathGet.

Methods:

- **long int MessageBytes() const;**
  Return the number of Message bytes set in the DcppBuffers variable.
- **long int MaxMessages() const;**
  Return the maximum number of messages set in the DcppBuffers variable.
- **long int ReplyBytes() const;**
  Return the number of Reply bytes set in the DcppBuffers variable.
- **long int MaxReplies() const;**
  Return the maximum number of replies bytes set in the DcppBuffers variable.

7.6 DcppTransaction — A type to maintain details of transactions initiated by a DcppTask object.

Derivation: This is a base type.

IncludeFile: dcpp.h

Description: This type is generally used internally to the class DcppTask and associated routines such as DcppDispatch. As a result it is not documented further here.

7.7 DcppTask — A class that provides access to the DRAMA messaging facilities.

Derivation: This is a base type.

Include File: dcpptask.h

Description: An object of this class provides an interface allowing you to send messages to other **DRAMA** tasks. At the most basic level, you provide the task name to an object constructor and then use the various methods to get a path to the task and send messages to it.

If a messaging routine returns `DcppReschedule`, then the action should reschedule to await completion messages, which should be processed using the **DcppDispatch()** routine or the **DcppHandler** class. You can provide callback routines that will be invoked in by these routines when a particular message completes.

The constructor and other methods allow you to optionally set details allowing the program to be loaded automatically on a Get Path operation. The communication buffer sizes can also be set.

These methods may be used from `UFACE` context by first invoking the routine **DcppUfaceCtxEnable()**.

Simple Types: The following types are defined and used in this and related classes (**DcppHandler** and **DcppMonitor**).

DcppVoidPnt A pointer to void

DcppVoidPntPnt A pointer to a pointer to void.

DcppHandlerRet An enumerated type with the following possible values - **DcppReschedule**, **DcppNotHandled** and **DcppFinished**.

Procedure Type: Several functions in this and related classes (**DcppHandler** and **DcppMonitor**) take procedures of the following types as arguments.

```
typedef DcppHandlerRet (*DcppHandlerRoutine)
                         DcppVoidPnt ClientData,
                         StatusType *status);
typedef void (*DcppBulkTransRoutine)
                        (unsigned long Transferred,
                         unsigned long Total,
                         DcppVoidPnt ClientData,
                         StatusType *status);
```

Constructors:

- **DcppTask(**
  const char * name,
  const char * node=0,
  const char * file=0);

Constructs an object to communicate with the drama task specified by "name" running on machine "node". If the task needs to be loaded, it will be loaded on machine "node" from filename "file". If "node" is not supplied, the machine this program is running is assumed and if "file" is not supplied, no load is done.

Operators: The assignment operator and copy constructor are private, preventing assignment and copying of items of this type, since these operations are not sensible for this type. No other operators are provided.

Methods 1: The following methods apply only if used prior to a GetPath operation.

- **void SetName(const char * TaskName);**
  Sets the name of the task to look for on a GetPath operation, or to load the program as if it is to be loaded.
- **void SetLocation(const char * Location);**
  Sets the node the task is running on or to be loaded on.
- **void SetFile(const char * File);**
  Sets the name of the file to load the program from if it needs to be loaded.
- **void SetBuffers(const DcppBuffers & Buffers);**
  Set the message buffer sizes to be used in communication with this task.
- **void SetFlowControl();**
  Enable flow control on this path.
- **void SetProcess(const char * ProcessName);**
  Set Process name to use when loading. See **DitsLoad(3)** for more details.
- **void SetArgument(const char * LoadArg, bool Append=false);**
  Set the arguments for the load operation. See **DitsLoad(3)** for more details.
- **void SetPriority(int Priority, bool Absolute=false);**
  Set the priority of the task if it is loaded. If "Absolute" is true, then we are setting the absolute priority, otherwise, the priority relative to the Imp master task doing the load operation.
- **void SetNames(bool Flag = true);**
  Set `DITS_M_NAMES` flag to **DitsLoad(3)** operation.
- **void SetSymbols(bool Flag = true);**
  Set `DITS_M_SYMBOLS` flag to **DitsLoad(3)** operation.
- **void SetProg(bool Flag = true);**
  Set `DITS_M_PROG` flag to **DitsLoad(3)** operation.
- **void LogLoad(bool Flag = true);**
  Enable or disabling logging of load operations. If enabled, messages are output using **MsgOut(3)** when loading occurs.
- **void ClearState();**
  Clears the internal state, use only if re-loading etc.

Methods 2: Getting and losing paths.

- **DccppHandlerRet GetPath(**
  StatusType * status,
  DcppHandlerRoutine SuccessHandler = 0,
  DcppHandlerRoutine ErrorHandler = 0,
  DcppVoidPnt ClientData = 0);

  The GetPath method will initiate getting a path to a task. If the GetPath is successfully initiated, then DcppReschedule will be returned and the caller should reschedule to await completion. Otherwise, DcppFinished will be returned and status set bad. If the task is unknown to the system and sufficient information is available, then an attempt will be made to load the task. Note, a successful GetPath operation will always result in a reschedule to ensure consistency. If the DcppDispatch routine or DcppHandle class is used to handle the reschedule, then the appropriate callback routines specified will be invoked.
- **void LosePath(StatusType * status);**

  The LosePath method loses a path and resets the state of the DcppTask object such that you can again find the path.

Methods 3: Informational

- **const char * TaskName() const;**

  Return a pointer to the name of the actual task. This is valid until a GetPath or SetName method is invoked.
- **bool GetPathLoaded() const ;**

  Returns true if the last GetPath operation loaded the task, false if the task was already running.
- **const char * GetArgument() const;**

  Returns a pointer to the current argument string, to be used in the next load operation. Valid until the SetArgument method is invoked.
- **const char * Location() const;**

  Returns a pointer to the current load location string, to be used in the next load operation. Valid until the SetLocation method is invoked.
- **void TaskLoggingOn();** Turns logging on for all DcppTask transactions for the specifie dtask. Simple messages are output using MsgOut each time a message event occurs.
- **void TaskLoggingOff();** Turns logging off for DcppTask transactions for the specified task. Note, `LoggingOn()` will override this.
- **static void LoggingOn();** Turns logging on for all DcppTask transactions and all tasks. Simple messages are output using MsgOut each time a message event occurs.
- **static void LoggingOff();** Turns logging of all tasks off. Task specified logging, set by `TaskLoggingOn()` may still operate.

Methods 4: Message operations

The following methods will initiate sending a message of the given type to the task. If the message is successfully initiated, then `DcppReschedule` will be returned. Otherwise status is set bad and `DcppFinished` is returned. If the **DcppDispatch()** routine or **DcppHandle** class is used to handle the reschedule, then the appropriate callback routines specified will be invoked. You can ignore the results of a message by specifying **DcppTask::DiscardResponse** as the `SuccessHandler` argument. In this case, `DcppFinished` is returned even if the message is initiated.

The "Forget" versions will immediately orphan the transaction. For these cases, if and only if there is a non-zero handler routine (any one), and if the orphaned transaction is taken over by an orphan handler, then the orphan handler can use **DcppDispatch()** to invoke the handlers in the normal way - i.e. the "Forget" versions can be used to pass control of the transaction to another action.

- **DcppHandlerRet Obey(**
  const char * Name,
  StatusType * status,
  const SdsId &Arg= SdsNull,
  DcppHandlerRoutine SuccessHandler=0,
  DcppHandlerRoutine ErrorHandler=0,
  DcppHandlerRoutine TriggerHandler=0,
  DcppVoidPnt ClientData = 0,
  DcppHandlerRoutine ErsHandler=0,
  DcppHandlerRoutine MsgHandler=0,
  DitsTransIdType * tid);

  Initiates an Obey message. "Arg" is the Sds Id of the argument to the action and SdsNull can be specified if no argument is required. "SuccessHandler" will be invoked when the initiated message completes successfully while "ErrorHandler" is invoked if it completes with an error. "TriggerHandler" is invoked for any trigger messages. "ClientData" is passed to each of the handler routines when they are invoked. "ErsHandler" is invoked for any ERS messages (`DitsGetEntReason()` has returned `DITS_REA_ERROR`) and "MsgHandler" for any `MsgOut()` messages (`DitsGetEntReason()` has returned DITS_REA_MESSAGE). Note that ERS and `MsgOut()` messages are also dependent on `DitsInterested()`. If this has not enabled handling of these messages, you won't get them. "tid" can be used to return the underlying transaction of of the standard transaction. This should only be used with "Spawable action" for passing to `DitsSpawnKickArg()`.

  If a particular handler is not supplied, an appropriate message is written to the user using **MsgOut(3)** if such an event occurs.

- **DcppHandlerRet Obey(**
  const char * Name,
  StatusType * status,
  const SdsId &Arg,
  DcppHandlerRoutine SuccessHandler,
  DcppHandlerRoutine ErrorHandler,

DcppHandlerRoutine TriggerHandler,
DcppVoidPnt ClientData,
DcppHandlerRoutine ErsHandler,
DcppHandlerRoutine MsgHandler,
SdsId * tidArg);

As per the pervious version execpt the the transaction id is returned already wraped up in an SdsId item. The original value of this SdsId item is deleted (with the structure deleted and id freeed if appropiate).

All this is doing is calling DcppSpawnKickArg(3) and returning the result from that rather then the transaction id itself.

Note that you get a different affect if `*tidArg` is `const`. See new next method.

- **DcppHandlerRet Obey(**
  const char * Name,
  StatusType * status,
  const SdsId &Arg,
  DcppHandlerRoutine SuccessHandler,
  DcppHandlerRoutine ErrorHandler,
  DcppHandlerRoutine TriggerHandler,
  DcppVoidPnt ClientData,
  DcppHandlerRoutine ErsHandler,
  DcppHandlerRoutine MsgHandler,
  const SdsId * tidArg);

  As per the original version execpt the the transaction id is returned already wraped up in an SdsId item. In this version (`*tidArg` is `const`, it is assumed that you are updating an item previously created by DcppSpawnKickArg(3)/DitsSpawnKickArg(3). See the previous version of Obey if you want to create a new Sds structure.

  All this is doing is calling DcppSpawnKickArgUpdate(3) and returning the result from that rather then the transaction id itself.

- **DcppHandlerRet Obey(**
  const char * Name,
  const DcppShared & SharedMem
  bool sds,
  int NotifyBytes,
  StatusType * status,
  DcppHandlerRoutine SuccessHandler=0,
  DcppHandlerRoutine ErrorHandler=0,
  DppBulkTransRoutine BulkTransHandler=0,
  DcppHandlerRoutine BulkDoneHandler=0,
  DcppHandlerRoutine TriggerHandler=0,
  DcppVoidPnt ClientData = 0,
  DcppHandlerRoutine ErsHandler=0,
  DcppHandlerRoutine MsgHandler=0,
  DitsTransIdType * tid);

  As per the original version of `Obey` except that this version is used to send bulk data. The shared memory describing the bulk data is specified by SharedMemInfo. See

the **DcppShared** class for details on how to set up such a structure. The flag `sds` should be set true if the bulk data contains an Exported SDS item. `NotifyBytes`, if non-zero specifies a number of bytes after the processing of which the target task will report progress. `BulkDoneHandler` is invoked when the target task has finished with and released the bulk data. `BulkTransHandler` is invoked to notify this task of the target tasks progress in processing the bulk data - normally but not necessarly at the rate determined by `NotifyBytes`. All other arguments are as per the original version of `Obey`.

Versions of this with the transaction id returned using an Sds structure are also available. Compare the first three versiosn of Obey to see the variations.

- **DcppHandlerRet Kick(**
  const char * Name,
  StatusType * status,
  const SdsId &Arg= SdsNull,
  DcppHandlerRoutine SuccessHandler=0,
  DcppHandlerRoutine ErrorHandler=0,
  DcppVoidPnt ClientData = 0,
  DcppHandlerRoutine ErsHandler=0,
  DcppHandlerRoutine MsgHandler=0);

  Initiates a Kick message.

- **DcppHandlerRet Kick(**
  const char * Name,
  const DcppShared & SharedMem
  bool sds,
  int NotifyBytes,
  StatusType * status,
  DcppHandlerRoutine SuccessHandler=0,
  DcppHandlerRoutine ErrorHandler=0,
  DppBulkTransRoutine BulkTransHandler=0,
  DcppHandlerRoutine BulkDoneHandler=0,
  DcppVoidPnt ClientData = 0,
  DcppHandlerRoutine ErsHandler=0,
  DcppHandlerRoutine MsgHandler=0);

  Initiates a bulk data Kick message. As per bulk data Obey message except that in this case the `Sucess` handler routine may be invoked before the `BulkDoneHandler`. This may occur if the target task has used **DitsBulkArgInfo(3)** to access the bulk data from its kick handler routine but does not release it (using **DitsBulkArgRelease(3)** until a later entry in its obey handler. You should assume either other is possible and if need be use **DitsGetEntComplete(3)** to determine which one indicates completion.

- **DcppHandlerRet Get(**
  const char * Name,
  StatusType * status,
  DcppHandlerRoutine SuccessHandler=0,
  DcppHandlerRoutine ErrorHandler=0,
  DcppVoidPnt ClientData = 0,

DcppHandlerRoutine ErsHandler=0);

Initiates a Get parameter message.

- **DcppHandlerRet MGet(**
  DcppHandlerRoutine SuccessHandler,
  DcppHandlerRoutine ErrorHandler,
  DcppVoidPnt ClientData,
  StatusType * status, unsigned count,
  [const char *name ...]);

  Initiates a multiple parmameter get message. You supply a count, which must be greater then zero and supply the same number of parameter names as the last arguments. Each name is a character string. The values of all the named parameters are returned.

- **DcppHandlerRet MGet(**
  DcppHandlerRoutine SuccessHandler,
  DcppHandlerRoutine ErrorHandler,
  DcppHandlerRoutine ErsHandler,
  DcppVoidPnt ClientData,
  StatusType * status, unsigned count,
  [const char *name ...]);

  Alternative MGet interface, which allows an ErsHandler to be specfied.

- **DcppHandlerRet Set(**
  const char * Name,
  StatusType * status,
  const SdsId & Arg= SdsNull,
  DcppHandlerRoutine SuccessHandler=0,
  DcppHandlerRoutine ErrorHandler=0,
  DcppVoidPnt ClientData = 0,
  DcppHandlerRoutine ErsHandler);

  Initiates a Set parameter message.

- **DcppHandlerRet Control(**
  const char * Name,
  StatusType * status,
  const SdsId & Arg= SdsNull,
  DcppHandlerRoutine SuccessHandler=0,
  DcppHandlerRoutine ErrorHandler=0,
  DcppHandlerRoutine TriggerHandler=0,
  DcppVoidPnt ClientData = 0,
  DcppHandlerRoutine ErsHandler=0);

  Initiates a Control message.

- **DcppHandlerRet Monitor(**
  const char * Name,
  StatusType * status,
  const SdsId & Arg= SdsNull,
  DcppHandlerRoutine SuccessHandler=0,

DcppHandlerRoutine ErrorHandler=0,
DcppHandlerRoutine TriggerHandler=0,
DcppVoidPnt ClientData = 0,
bool SendCurrent=0,
DcppHandlerRoutine ErsHandler=0);

Initiates a Monitor message. If "SendCurrent" is set true, the current values of the parameters being monitored are sent immediately.

- **void ObeyForget(**
  const char * Name,
  StatusType * status,
  const SdsId & Arg= SdsNull,
  DcppHandlerRoutine SuccessHandler=0,
  DcppHandlerRoutine ErrorHandler=0,
  DcppHandlerRoutine TriggerHandler=0,
  DcppVoidPnt ClientData = 0,
  DcppHandlerRoutine ErsHandler=0,
  DcppHandlerRoutine MsgHandler=0);

  Initiates but forgets an Obey message.

- **void KickForget(**
  const char * Name,
  StatusType * status,
  const SdsId & Arg= SdsNull,
  DcppHandlerRoutine SuccessHandler=0,
  DcppHandlerRoutine ErrorHandler=0,
  DcppVoidPnt ClientData = 0,
  DcppHandlerRoutine ErsHandler=0,
  DcppHandlerRoutine MsgHandler=0);

  Initiates but forgets a Kick message.

- **void MonitorForget(**
  const char * Name,
  StatusType * status,
  const SdsId & Arg= SdsNull,
  DcppHandlerRoutine SuccessHandler=0,
  DcppHandlerRoutine ErrorHandler=0,
  DcppHandlerRoutine TriggerHandler=0,
  DcppVoidPnt ClientData = 0,
  bool SendCurrent=0,
  DcppHandlerRoutine ErsHandler = 0);

  Initiates but forgets a Monitor message. If "SendCurrent" is set true, the current values of the parameters being monitored are sent immediately.

Methods 4: Miscellaneous methods

- **DcppHandlerRet Handle(**
  DcppTransaction * Transaction,
  StatusType * status);

When the message methods start a transaction, they associate the address of a **DcppTransaction** object with the transaction. This address can be retrieved with **DitsGetTransData(3)** when related messages are received. This method is used by the **DcppDispatch()** routine or the **DcppHandler** class to handle the responses to messages sent using the above functions. It is not normally invoked directly by user code.

- **static DcppHandlerRet DiscardResponse(**
  DcppVoidPnt ClientData,
  StatusType * status);

  This method is not invoked directly. When specified as a success handler to one of the above functions, any response to the message is ignored.

7.8 DcppHandler — A class that completely hides the details of the rescheduling required to handle messages started by DcppTask object methods.

Derivation: This is a base type.

Include File: dcpphandler.h

Description: When a **DcppHandler** object is installed for the current action, it calls **DitsPutObeyHandler(3)** supplying its own routine to handle future reschedules of the action. This will result in the various handlers specified in calls to **DcppTask** messaging methods will be invoked when appropriate events occur. Such handler routines should return **DcppReschedule**, to indicate that more messages are expected and rescheduling should occur. Otherwise, they should return **DcppFinished**.

By default, a **DcppHandler** will keep track of one thread of messages. If you wish to have multiple threads active at one time you should indicate to the **DcppHandler** object that there are multiple threads. For each thread after the first, you should invoke the **DcppHandler NewThread** method. [2]. The **DcppHandler** object will then continue to reschedule the action until all threads have returned DcppFinished. To allow consistency in calls, you may specify the optional threads argument to Install as 0, which means you must call the increment operator for every thread.

In addition, you can specify routines to be invoked when the last reschedule has completed or when rescheduling errors occur or when a timeout occurs. Again, by returning **DcppReschedule**, they indicate that rescheduling should occur. Otherwise, the should return **DcppFinished**. (You can use these to manage multiple threads yourself if desired, by returning **DcppReschedule** if there are other threads outstanding.

When dealing with multiple threads the finished handler is called after all threads have completed. By default, the error handler will be called instead of the finished handler if any thread completed with an error. The user may invoke ErrorStatus to get the status associated with the first error while subsequent errors are reported using ErsRep. It is possible for the error handler to be invoked once for every thread which completes with an error. This is enabled by calling SetMultiCallErrorMode. In this case, if the last thread to complete completes without an error, then the finished handler will be invoked. User success handlers can check if a previous thread completed with an error by getting the error status. This will be non-zero if a thread has completed with an error (except when SetMultiCallErrorMode has been invoked).

It should be noted that it would be unusual to declare a DcppHandler variable on the stack. Normally they are either static items or dynamically allocated. In the later case, they can be free by the finished handler or error handler if these handlers are returning **DcppFinished**. Note for error handles, this is not the case if the multiple error mode is enabled, see above.

Constuctors:

[2]Previously, the increment operator (either pre or post increment) was used for this. It is currently still available but will be removed at some stage

- **DcppHandler(**
  DcppHandlerRoutine FinishedHandler = 0,
  DcppHandlerRoutine ErrorHandler = 0,
  DcppVoidPnt ClientData = 0,
  float Timeout = -1);

  Creates a DcppHandler type, specifying the routine to be invoked when rescheduling completes ("FinishedHandler"), a routine to invoke when an error occurs whilst rescheduling ("ErrorHandler"), "ClientData" to pass to those items and a timeout between reschedules.

Operators: The assignment operator and copy constructors are private, preventing assignment and copying of items of this type, since these operations are not sensible for this type.

Post and pre increment operations are provided to increment the count of outstanding threads but have been replaced by the **NewThread** method and will be removed at some stage. They return void.

Methods:

- **void Install(StatusType * status, int threads = 1);**
  Install installs a constructed handler. This will cause the Obey handler and ActData to be overwritten for the invoking action to be overridden.

  If you have set a timeout, using **SetTimeout** below, before this point, then Install will enable it. You must put an appropriate request yourself. i.e. `DITS_REQ_MESSAGE`.

  To allow consistency in calls, you may specify the optional threads argument to Install as 0, which means you must call the increment operator for every thread you start.

  If you need to access the handler from action routines which don't have access to this data item, such as a Kick routine, you can access the handler using DitsGetActData(). Cast the value returned from DitsGetActData() to the type (DcppHandler *) and access the handler using this pointer value.
- **void DeInstall(StatusType * status);**
  DeInstall will restore ActData to it's value when **Install** as invoked.
- **void SetFinished(DcppHandlerRoutine FinishedHandler);**
  Set the routine to be invoked when rescheduling is complete.
- **void SetError(DcppHandlerRoutine ErrorHandler);**
  Set the routine to be invoked if an error occurs during rescheduling.
- **void SetData(DcppVoidPnt ClientData);**
  Set the client data item to be passed to the handler routines.
- **void SetTimeout(float Timeout);**
  Set the reschedule timeout value. Does not changes the Timeout Handler.
- **void SetTimeout(DcppHandlerRoutine TimeoutHandler, float Timeout);**
  Set the reschedule timeout value and a handler to be invoked when the timeout occurs. If this handler returns DcppReschedule, then the action is reschedule as if nothing had happened. If it returns DcppFinished, then the action completes immediately. If

no timeout handler has been enabled, then a timeout is considered as an error, with status `DITS__APP_TIMEOUT`.

- **void SetMultiCallErrorMode(int enabled = 1);** If enable, then the user error handler is invoked when ever any thread completes with an error. By default, it will only be invoked after the last thread completes. If no use error handler is supplied, then enabling this causes the action to complete when an error occurs instead of waiting for all threads to complete.
- **StatusType ErrorStatus() const;**
  This method should only be invoked in the error handler. It returns the status which caused the error handler to be invoked. If invoked in other places, the returned value if undefined.
- **DcppVoidPnt GetData() const;**
  This method returns the ClientData item set above.
- **void NewThread();**
  Increment the count of the number of threads of messages being managed by this DcppHandler object.

7.9 DcppMonitor — A class which supports parameter monitoring.

Derivation: This is a base type.

Include File: DcppMonitor.h

Description: This class provides a wrap around to parameter monitoring operations. It uses a **DcppTask** object previously constructed by the user and on which a **GetPath** operation must have been completed.

The user sets up a monitoring transaction specifying the parameters to be monitored and routines to be invoked when monitor event occurs. Various methods allow you to start normal and forward monitoring, to Add and Delete parameters and to Cancel monitoring. The action invoking the Monitor and Forward methods should reschedule to await responses, using the **DcppDispatch()** routine or **DcppHandler** class to handle the messages.

The "Forget" versions will immediately orphan the transaction. If and only if there is a non-null handler routine (any one), and then if the orphaned transaction is taken over by an orphan handler, then the orphan handler can use **DcppDispatch()** to invoke the handlers in the normal way - i.e. the "Forget" versions can be used to pass control of the transaction to another action.

Procedure Type: The following procedure type is used for the routine that is invoked when a parameter changed message is received. When the routine is invoke, the name argument contains the name of the parameter that has changed. The type argument contains the Sds type tag while the value argument, if non-zero, is a pointer to the value of the parameter (in the appropriate type). If value is 0, then the type is non-scalar and you should use DitsGetArgument to get and interpret the parameter value. If the type is `ARG_STRING` then the value points to a character string.

```
typedef void (*DcppMonChangedRoutine)
                const   char * name,
                SdsCodeType type,
                DcppVoidPnt value,
                DcppVoidPnt ClientData,
                StatusType * status);
```

Constructors:

- **DcppMonitor(DcppTask * Task);**
  Constructs a **DcppMonitor** object specifying a task for which a **DcppTask GetPath** operation must be done before any **DcppMonitor** operation can be performed.

Operators: The assignment operator and copy constructor are private, preventing assignment and copying of items of this type, since these operations are not sensible for this type. No other operators are provided.

Methods:

- **DcppTask * Task() const;**

  Returns a pointer to the **DcppTask** object which has been associated with this **DcppMonitor** object.

- **void Monitor(**
  DcppMonChangedRoutine ChangedHandler,
  DcppHandlerRoutine CompletedHandler,
  DcppVoidPnt ClientData,
  bool SendCurrent,
  int count, StatusType * status,
  [const char *Parameter ...]);

  The Monitor method results in monitor transactions being initiated. You can supply routines to be invoked when the parameter changes or the monitor completes, both of which are supplied with the specified `ClientData` item. If the `SendCurrent` boolean is true, the current values of the parameters are sent immediately.

  An optional list of parameter names is supplied. You must specify the number using the count argument. If you specify a count of -1, then only the first optional parameter is used, but it contains a space separated list of names.

  You should reschedule the invoking action and use either **DcppDispatch()** or **DcppHandler** to handle the reschedules which will result from parameter changed or monitor completion messages.

- **void Forward(**
  const char * Task,
  const char * Action,
  DcppHandlerRoutine CompletedHandler,
  DcppVoidPnt ClientData,
  bool SendCurrent,
  int Count,
  StatusType * status,
  [const char * Parameter ...]);

  The Forward method results in forward monitor transactions being initiated to the specified `Task`, using the specified `Action`.

  You can supply routine a routine to be invoked when the Monitor completes.

  If the `SendCurrent` boolean is true, the current values of the parameters are sent immediately.

  An optional list of parameter names is supplied. You must specify the number using the count argument. If you specify a count of -1, then only the first optional parameter is used, but it contains a space separated list of names.

  You should reschedule the invoking action and use either **DcppDispatch()** or **DcppHandler** to handle the reschedules which will result from monitor completion messages.

- **void Forward(**
  const char * Task,
  const char * Action,
  DcppHandlerRoutine StartedHandler,

DcppHandlerRoutine CompletedHandler,
DcppVoidPnt ClientData,
bool SendCurrent,
int Count,
StatusType * status,
[const char * Parameter ...]);

As per the previous version of the **Forward**, but in this version, you may specify the `StartedHandler` parameter, a routine to be invoked when DcppMonitor receives notification that the monitor operation has started sucessfully.

- **void MonitorForget(**
  DcppMonChangedRoutine ChangedHandler,
  DcppHandlerRoutine CompletedHandler,
  DcppVoidPnt ClientData,
  bool SendCurrent,
  int Count,
  StatusType * status,
  [const char * Parameter ...]);

  As per the **Forward** method, but it will immediately orphan the transaction. If and only if there is a non-null **ChangedHandler** or **CompletedHandler** and then if the orphaned transaction is taken over by an orphan handler, then the orphan handler can use **DcppDispatch()** to invoke the handlers in the normal way - i.e.
- **void ForwardForget(**
  const char * Task,
  const char * Action,
  DcppHandlerRoutine CompletedHandler,
  bool SendCurrent,
  DcppVoidPnt ClientData,
  int Count,
  StatusType * status,
  [const char * Parameter ...]);

  As per the **Forward** method, but it will immediately orphan the transaction. If and only if there is a non-null **CompletedHandler** and then if the orphaned transaction is taken over by an orphan handler, then the orphan handler can use **DcppDispatch()** to invoke the handlers in the normal way - i.e.
- **void Add(**
  const char * Parameter,
  StatusType * status,
  DcppHandlerRoutine SuccessHandler = 0,
  DcppHandlerRoutine ErrorHandler=0,
  DcppVoidPnt ClientData=0);

  Adds a new parameter to the list of parameters to be monitored by this Monitor object. You must have already used one of the above methods to setup a monitor.

  This method need not be invoked in the same action which initiated the monitor, but you should reschedule the invoking action this call and use either **DcppDispatch()** or **DcppHandler** to handle the completion or error messages.

- **void Delete(**
  const char * Parameter,
  StatusType * status,
  DcppHandlerRoutine SuccessHandler = 0,
  DcppHandlerRoutine ErrorHandler=0,
  DcppVoidPnt ClientData=0);

  Delete a parameter from the list of parameters to be monitored by this Monitor object. You must have already used one of the above methods to setup a monitor.

  This method need not be invoked in the same action which initiated the monitor, but you should reschedule the invoking action this call and use either **DcppDispatch()** or **DcppHandler** to handle the completion or error messages.

- **void Cancel(**
  StatusType * status,
  DcppHandlerRoutine SuccessHandler = 0,
  DcppHandlerRoutine ErrorHandler=0,
  DcppVoidPnt ClientData=0);

  Cancel this monitor. This will result in the calling of `CompletionHandler`, if any, specified when the monitor was started. (This will occurs after rescheduling of the invoking action). Once the completion handler has been invoked, you can restart monitoring using an appropriate routine.

  This method need not be invoked in the same action which initiated the monitor, but you should reschedule the invoking action this call and use either **DcppDispatch()** or **DcppHandler** to handle the completion or error messages.

7.10 DcppShared — A class that provides a C++ interface to creation of shared memory segements for bulk data..

Derivation: Arg←SdsId

Include File: dcpp.h

Description: This class wraps up management of the shared memory segements used for bulk data operations.

Constructors:

- **DcppShared(**
  long Size,
  StatusType * status,
  SharedType Type = Create,
  const char *Name = "",
  int Key = 0,
  void * Address = 0);

  Create a shared memory segment of `Size` bytes. The Segement type is specified by `Type`, which has the following possible values

  Create A temporary shared memory segement in some form suitable for the system on which the program is running. The

  Create, `Name`, `Key` and `Address` arguments will be ignored.

  Gblsec (VMS Only) Create a VMS global page section.

  Name should be the name for the global section. The `Key` and `Address` arguments will be ignored.

  Global (VxWorks Only) The mapped section is just a section of memory, starting at a specified address. `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.

  ShMem (Unix Only). The mapped section can be created as System V shared memory. `Key` specifies the identifier for the shared memory and `Name` should be a Null string. `Address` is ignored. If `Create` is true, the section is created.

  MMap (Unix Only). The mapped section can be created as a file accessed through **mmap()**. `Name` should be the full name of the file, and `Key` is ignored. `Address` is ignored. If `Create` is true, the file is created.

  See **DitsDefineShared(3)** for more information but note that the `Address` argument in this method is input only, whilst in **DitsDefineShared(3)** it is both an input and output argument.

- **DcppShared(**
  const SdsId & Template,
  StatusType * status,
  SharedType Type = Create,
  const char *Name = "",

int Key = 0,
void * Address = 0);

Create a shared memory segment of containing an Sds structure based on the `Template` SDS structure. The size memory is required is determined before the segement is created in a similar way to the previous structure. The `Template` SDS structure is then exported into the shared memory segement.

See the previous contructor for details about the other arguments to this constructor.

- **DcppShared**
  DitsSharedMemInfoType *Info
  void *Address = 0,
  SdsIdType ID = 0,
  bool Free = false);

  This constructor initializes the object from an existing `DitsSharedMemInfoType` structure specified by the `Info` variable which has been set up by calling **DitsDefineShared(3)** (or **DitsDefineSdsShared(3)**

  The `Address` variable specifies the address of the shared memory as returned by **DitsDefineShared(3)**. `ID` is the SDS of any SDS structured which has been exported into the shared memory. If zero, it is assumed there is no such structure.

  Note for when using the **GetAddress()** method, below. The address can only be known if passed in usingthe constructor. If it was not set, then **GetAddress()** will set status bad.

  The SDS ID (if any) and shared memory segement are only released when by this object's destructure if the `Free` flag is set true.

  The default arguments to this constructor allow an object of type `DitsSharedMemInfoType *` to be passed where ever an object of type `DcppShared` is required.

Operators: The assignment operator and copy constructor are private, preventing assignment and copying of items of this type, since these operations are not sensible for this type.

No other operators are provided.

Methods:

- **void GetInfo (**
  const DitsSharedMemInfoType * * Info,
  StatusType * status) const ;
- **void GetInfo (**
  DitsSharedMemInfoType * * Info,
  StatusType * status);

  Returns a pointer to the underlying `DitsSharedMemInfoType` variable used by this object to represent the shared memory. The only appropiate use of this is to pass the shared memory details to message sending routines.

  `const` and non-`const` versions.
- **void GetAddress (**
  const DitsSharedMemInfoType * * Info,
  StatusType * status) const ;

- **void GetAddress (**
  DitsSharedMemInfoType * * Info,
  StatusType * status);

  Returns the address of the shared memory associated with this object.

  This will set status bad if the object was created with the constructor which takes the address of a variable of type `DitsSharedMemInfoType` and the `Address` argument was not specified. (Status will be set to `DCPP__SHARENOADD`

  `const` and non-`const` versions.

- **const SdsId & GetSds () const ;**
- **SdsId & GetSds ();**

  Return a reference to the SDS ID representing the structure, if any, exported into the shared memory segement. If no such structure exists, this object will be an invalid SDS ID which will cause SDS operations to set status bad.

  `const` and non-`const` versions.

8 Routines

This section documents some related routines, which don't belong in a class.

8.1 DcppDispatch — Dispatches reschedule messages to handlers.

Function: Dispatches reschedule messages to handlers if they were the result of transactions started using the **DcppTask** class.

Description: **DcppDispatch()** should be invoked by action handler routines to handle reschedules caused by messages sent by **DcppTask** objects. It will send an appropriate message to the task object involved.

By checking the return value, you can determine if the message was handled and, if so, if another reschedule is expected.

Call:

(DcppHandlerRet) = DcppDispatch (status)

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

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

Return Value:

`DcppReschedule`	The invoking action should reschedule to await more messages.
`DcppNotHandled`	The message that causes this reentry was not initiated by a DcppTask object.
`DcppFinished`	No more reschedules expected in the context of whatever causes the reschedule.

Include file: Dcpp.h

8.2 DcppUfaceCtxEnable — Enables use of the DcppTask methods from within `UFACE` context.

Function: Enables use of the **DcppTask** methods from within `UFACE` context.

Description: When using DcppTask objects, this call replaces the normal use of **DitsUfaceCtxEnable(3)**. Please see the details of that routine for a proper explanation of `UFACE` context and where this routine should be invoked.

This function simply casts **DcppDispatch()** to the type `DitsActionRoutineType` and passes it to **DitsUfaceCtxEnable(3)**. The result is that handlers for any following calls to **DcppTask** objects are correctly invoked.

Call:
(void) = DcppUfaceCtxEnable (status)

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

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

Include file: Dcpp.h

8.3 DcppSpawnKickArg — Create an argument structure used when kick actions which spawn.

Function: Create an argument structure used when kick actions which spawn

Description: This is a C++ interface to **DitsSpawnKickArg(3)**.

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(3)**) 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 Arg/SdsId methods. Also, it is possible to change the transaction id in this structure using **DcppSpawnKickArgUpdate(3)**.

Call:
(void) = DcppSpawnKickArg (transid, arg, status)

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

(`>`) transid (DitsTransIdType) The transaction id returned when the transaction was started.

(`<`) arg (SdsId `*`) The SdsId item to use. The original value of this is deleted (with the ID free and structure deleted if necessary) before the item created here is copied in.

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

Include file: Dcpp.h

8.4 DcppSpawnKickArgUpdate — Update an argument structure used when kick actions which spawn.

Function: Update an argument structure used when kick actions which spawn

Description: This is a C++ interface to **DitsSpawnKickArgUpdate(3)**.

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(3)**) 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 update the later structure with a new transaction id.

Call:

(void) = DcppSpawnKickArgUpdate (transid, arg, status)

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

(>) transid (DitsTransIdType) The transaction id returned when the transaction was started.

(!) arg (SdsId *) The SdsId item to update.

(!) status (StatusType *) Modified status.

Include file: Dcpp.h

References

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

[2] Tony Farrell, AAO. *23-Feb-1995, Distributed Instrumentation Tasking System.* Anglo-Australian Observatory **DRAMA** Software Document 5.

[3] Jeremy Bailey , AAO. *6-Aug-1993, Self-defining Data System.* Anglo-Australian Observatory **DRAMA** Software Document 7.

[4] Tony Farrell, AAO. *18-Jan-1994, Generic Instrumentation Task Specification.* Anglo-Australian Observatory **DRAMA** Software Document 9.

A A more complex example

Below is a rewrite of the program "`ctest.c`", described in [1], using the C++ tasking classes. This program sends multiple actions to to subsidiary tasks (`TEA` and `COFFEE`) and awaits fro the responses.

```
/*
 *   c p p C t e s t . C
 *
 *  A re-implementation of "ctest.c" using the C++ interface to DRAMA.
 *
 *  Copyright (c) Anglo-Australian Telescope Board, 1995.
 *  Not to be used for commercial purposes without AATB permission.
 *
 *     @(#) $Id$
 *
 */
a
/*
 * Default message buffer sizes.  See DitsInit and DitsGetPath
 */
#define  DBUFSIZE 30000
#define  MESSAGEBYTES 400
#define  MAXMESSAGES 5
#define  REPLYBYTES 800
#define  MAXREPLIES 12

/*
 * Some configuration constants
 */
#define TEAMAX 5            /* Max number of actions to start in TEA    */
#define COFFEEMAX 5         /* Max number of actions to start in COFFEE */
#define TIMEOUT 0           /* Timeout in seconds to wait for messages  */
#define TEAARG 10           /* Argument to LapSang3 action to Tea task  */

/*
 * Include files.
 */
#include "status.h"
#include "DitsTypes.h"
#include "DitsSys.h"
#include "DitsFix.h"
#include "DitsUtil.h"
#include "dcpptask.h"
#include "dcpphandler.h"
#include "Ers.h"
```

```
#include "DitsMsgOut.h"

/*
 *  Action Handler routines.
 */
static void CTestFindPaths(StatusType * const status);
static void CTestExit(StatusType * const status);

/*
 *  The Task structures for each task we will control.
 */
static DcppTask tea("TEA");
static DcppTask coffee("COFFEE");


/*
 *  Main routine.  Just calls cppCteest.
 */
extern int cppCtest();
#ifdef DITS_MAIN_NEEDED
    extern int main()
    {
        return(cppCtest());
    }
#endif

/*
 *  Actual cppCtest main routine.
 */
extern int cppCtest()
{
    static int BufSize = DBUFSIZE;      /* Message buffer size      */
/*
 *  ActionMap associates action names with routines.  ActionMapSize is the
 *  size of the array.
 */
    static DitsActionMapType ActionMap[] = {
        {CTestFindPaths, 0, 0, "BREW" },
        {CTestExit, 0, 0, "EXIT" }
    };
    static int ActionMapSize = DitsNumber(ActionMap);
    StatusType status = STATUS__OK;

    DitsInit("CTEST",BufSize,0,&status);
    DitsPutActionHandlers(ActionMapSize,ActionMap,&status);
    DitsMainLoop(&status);
```

```
    return(DitsStop("CTEST",&status));

}

/*
 *  CTestExit responds to the exit action.  It sends exit commands to the TEA
 *  and COFFEE tasks and puts an exit request, causing this task to exit.
 *
 *  Note that we ignore an error from the Obey's to simplify code if the
 *  BREW has not been initiated and we discard any response so we can
 *  exit immediately.
 */
static void CTestExit(StatusType * const status)
{
    StatusType ignore = STATUS__OK;
    tea.Obey("EXIT",&ignore,0,DcppTask::DiscardResponse);
    ignore = STATUS__OK;
    coffee.Obey("EXIT",&ignore,0,DcppTask::DiscardResponse);
    DitsPutRequest(DITS_REQ_EXIT,status);
}

/*
 * Variables used accros the functions which implement the BREW action.
 */
static DcppHandler BrewHandler; /* A Reschedule handler for BREW        */
static int tea_active;          /* Number of actions active to the TEA task */
static int coffee_active;       /* Number of actions active to the COFFEE task*/
static int paths_active;        /* Number of Get Path operations active */
static int first;               /* First time flag                      */

/*
 *  Function prototypes.
 */
static DcppHandlerRet PathsFound(DcppVoidPnt ClientData,
                                 StatusType * const status);
static DcppHandlerRet ActionSuccess(DcppVoidPnt ClientData,
                                 StatusType * const status);
static DcppHandlerRet ActionError(DcppVoidPnt ClientData,
                                 StatusType * const status);
static DcppHandlerRet ActionTrigger(DcppVoidPnt ClientData,
                                 StatusType * const status);

/*
 *
 *  CTestFindPaths is the routine invoked on the first entry of the
 *  BREW action.  It initiates getting the paths to the other tasks.
```

```
 */
static void CTestFindPaths(StatusType * const status)
{
    if (!StatusOkP(status)) return;

/*
 *  Install the handler which will take control of rescheduling this
 *  action.
 */
    if (TIMEOUT > 0)
        BrewHandler.SetTimeout(TIMEOUT);
    BrewHandler.Install(status);

/*
 *  Set task buffer sizes.
 */
    tea.SetBuffers(DcppBuffers(MESSAGEBYTES,MAXMESSAGES,REPLYBYTES,MAXREPLIES));
    coffee.SetBuffers(DcppBuffers(MESSAGEBYTES,MAXMESSAGES,REPLYBYTES,
                                                            MAXREPLIES));
/*
 *  Get the paths to the tasks and reschedule.  PathsFound() will be invoked
 *  when the paths is found.
 */
    tea.GetPath(status,PathsFound);
    coffee.GetPath(status,PathsFound);
    paths_active = 2;
    DitsPutRequest(DITS_REQ_MESSAGE,status);

    if (*status != STATUS__OK)
        ErsRep(0,status,"Error trying to get paths to TEA and COFFEE");
}

/*
 * We get here when a path is found.
 */
static DcppHandlerRet PathsFound(DcppVoidPnt ClientData,
                                 StatusType * const status)
{
    if (*status != STATUS__OK) return(DcppFinished);
/*
 *  If there is still a GetPath outstanding, then we must reschdule again,
 *  otherwise, we can start the obey's.
 */
    --paths_active;
     if (!paths_active)
     {
```

```
        register i;
/*
 *      Clear global flags.
 */
        tea_active = 0;
        coffee_active = 0;
        first = 0;

/*
 *      Start COFFEEMAX actions with names of the form MOCHAn when n is a
 *      number range 1 to COFFEEMAX in task COFFEE
 */
        for (i = 1; (i <= COFFEEMAX) && StatusOkP(status); ++i)
        {
            char name[DITS_C_NAMELEN];
            sprintf(name,"MOCHA%d",i);
            if (coffee.Obey(name,status,0,ActionSuccess,ActionError,
                        ActionTrigger,DcppVoidPnt(&coffee)) == DcppReschedule)
                ++coffee_active;
            else
            {
                ErsRep(0,status,"failed to start action %s in task COFFEE - %s",
                                        name,DitsErrorText(*status));
            }
        }

/*
 *      Start TEAMAX actions with names of the form LAPSANGn when n is a
 *      number in  the range 1 to TEAMAX in task TEA.
 */
        for (i = 1; (i <= TEAMAX) && StatusOkP(status) ; ++i)
        {
            char name[DITS_C_NAMELEN];
            sprintf(name,"LAPSANG%d",i);
            if (i == 3)
            {
                Arg id(true,status);
                id.Put("COUNT",(short)TEAARG,status);
                tea.Obey(name,status,id,ActionSuccess,ActionError,
                                        ActionTrigger,DcppVoidPnt(&tea));
            }
            else
                tea.Obey(name,status,0,ActionSuccess,ActionError,
                                        ActionTrigger,DcppVoidPnt(&tea));
            if (*status == STATUS__OK)
                ++tea_active;
```

```
            else
            {
                ErsRep(0,status,"failed to start action %s in task COFFEE - %s",
                                        name,DitsErrorText(*status));
            }
        }

/*
 *      IF we have any active actions, then reschedule to await.
 *      completion.  Either ActionSuccess or ActionError will be invoked next
 */
        if ((coffee_active || tea_active)&&(*status != STATUS__OK))
        {
             ErsFlush(status);
        }
    }
    return (*status == STATUS__OK ? DcppReschedule : DcppFinished);
}


/*
 *  Invoked when an action completes successfully.
 */

static DcppHandlerRet ActionSuccess(DcppVoidPnt ClientData,
                                    StatusType * const status)
{
    DcppTask *task = (DcppTask *)ClientData;
/*
 *  We have a completion message.  Get information on the entry so we
 *   can work out which task.
 */
    if (task == &coffee)
        --coffee_active;
    else if (task == &tea)
        --tea_active;
    else
        ErsOut(0,status,"Action completion from unknown task");

/*
 *  First this through, send a KICK to TEA.
 */
    if (first)
    {
        if (tea.Kick("LAPSANG5",status,0,ActionSuccess,ActionError,
                                        DcppVoidPnt(&tea)) == DcppReschedule)
```

```
        {
            ++tea_active;
            first = 0;
        }
    }
/*
 *  If any actions still active, then reschedule.
 */
    if ((*status == STATUS__OK)&&(tea_active || coffee_active))
        return DcppReschedule;
    else
        return DcppFinished;
}
static DcppHandlerRet ActionError(DcppVoidPnt ClientData,
                                  StatusType * const status)

{
    DcppTask *task = (DcppTask *)ClientData;
/*
 *  We have an error completion message.  Get information on the entry so we
 *   can work out which task.  *status is the error status.
 */
    if (task == &coffee)
    {
        --coffee_active;
        ErsOut(0,status,"Error completion from task COFFEE - %s",
                                DitsErrorText(*status));
    }
    else if (task == &tea)
    {
        --tea_active;
        ErsOut(0,status,"Error completion from task TEA - %s",
                                DitsErrorText(*status));
    }
    else
    {
        ErsOut(0,status,"Error completion from unknown task - %s",
                                DitsErrorText(*status));
    }
/*
 *  Status should now be clear.  Reschedule if there is anything more
 *  to wait for.
 */
    if (tea_active || coffee_active)
        return DcppReschedule;
```

```
    else
        return DcppFinished;
}

/*
 *  Handle trigger messages.
 */
static DcppHandlerRet ActionTrigger(DcppVoidPnt ClientData,
                                    StatusType * const status)
{
    MsgOut(status,"Trigger message received");
    return DcppReschedule;
}
```