ASC Data Model Subroutine Interface: ANSI C user guide
Release 1.7

Jonathan McDowell, Michael S. Noble, Oliver Oberdorf, Scott Randall

March 30, 1998

Contents

1 Overview

1.1 FITS and QPOE

The Data Model fosters an abstract view of astronomical data files and provides data I/O transparently to FITS, QPOE and IMH format files. Access to these distinct formats is achieved by isolating and abstracting the format-specific calls into different I/O "kernels". The two file kernels currently supported are FITS and IRAF. To the Data Model, every 'dataset' is a series of 'blocks'. In FITS, a dataset is a file and a block is usually an HDU (Header Data Unit) which can be either a table or an image. In IRAF, a dataset is a directory containing PROS-type QPOE files and IMH files. Each block is either a QPOE event list (interpreted as a table) or an IMH image.

Note that as of DataModel Release 1.5, any single QPOE or IMH file may be opened for reading as a read-only dmDataset equivalent consisting of one table or image dmBlock.

Each block consists of header and data. In a table type block, the data is organized in columns, each with the same number of rows. Both header keys and table columns have a lot in common, and in FITS it is an emerging convention that one may regard header keys as columns which have the same value in each row. In the Data Model, we consider header keys and table columns as examples of a more generic object called a Descriptor. A Descriptor is so called because it contains the descriptive information associated with the raw data value or values; this descriptive information can be names, units, data type, etc. The data in an image is a special case of a column descriptor. There are two other kinds of descriptors: coordinate descriptors and filter descriptors. Subspace descriptors store information about how the main data has been filtered. Coordinate descriptors are 'virtual' column descriptors for which the data values are defined implicitly, in terms of a function of another descriptor. Image Axis Group descriptors, which carry information about the axes of an image, are a special case of coordinate descriptor.

1.2 A simple example

Here is a simple example of code which reads two tables and writes a third. In the GTI table, we open column descriptors by explicit column numbers since we know that different implementations of GTI tables use a variety of names for the columns but the data is always in columns 1 and 2. We use the GetScalars command to read each column at one gulp, since we know the GTI is probably small and we won't take much of a buffering hit. In the event table, we open column descriptors by name, allowing the possibility that the order of the columns may be moved around. Since the number of rows may be large, we read the data row by row to avoid FITSIO buffering problems.

The header key read returns a descriptor, which we can use to find out the key's unit or other properties, but most often we use it just to test that it is non-null, i.e. that the keyword is present.

Error checking is omitted from the code below for brevity, as are comments since the code is described above in the text.

```
dmDataset* input_ds;
dmDataset* output_ds;
dmBlock* gti_table;
dmBlock* event_table;
dmBlock* out_table;
dmDescriptor* start_col;
dmDescriptor* stop_col;
dmDescriptor* pha_col;
dmDescriptor* status_col;
double* start;
double* stop;
double livetime;
int spectrum[MAXPHA+1];
int pha;
short status;
int ngti;
int n;
int itzero;
double tzero;

input_ds  = dmDatasetOpen( "bas.fits" );
gti_table = dmBlockOpen( input_ds, "STDGTI" );

start_col = dmTableOpenColumnNo(gti_table,1);
stop_col  = dmTableOpenColumnNo(gti_table,2);
ngti      = dmTableGetNoRows(gti_table);

start = (double*)malloc(ngti*sizeof(double));
stop  = (double*)malloc(ngti*sizeof(double));

dmGetScalars_d(start_col, start, 1, ngti);
dmGetScalars_d(stop_col, stop, 1, ngti);
dmBlockClose(gti_table);

livetime = 0.0; for ( i= 0;i<ngti;i++ ) { livetime += stop[i] - start[i]; }
event_table=dmBlockOpen(input_ds, "STDEVT");

/* Look for any of various MJDREF keywords */
if( dmKeyRead_i( event_table, "MJDREFI", &itzero ) )
{
```

```
   (void)dmKeyRead_d( event_table, "MJDREFF", &tzero );
    tzero += itzero;
}
else if( !dmKeyRead_d( event_table, "MJDREF", &tzero ) )
   if ( dmKeyRead_i( event_table, "XS-MJDRD", &itzero ) )
   {
      (void)dmKeyRead_d( event_table, "XS-MJDRF", tzero );
       tzero += itzero;
   }
   else
   {
       tzero = 0.0;
       printf( "No MJDREF keyword found\n" );
   }

n = dmTableGetNoRows( event_table );
pha_col = dmTableOpenColumn( event_table, "PHA" );
status_col=dmTableOpenColumn( event_table, "STATUS" );
for ( row = 0; row < n; row++ )
{
    pha = dmGetScalar_l( pha_col );
    status = dmGetScalar_s( status_col );
    if ( status == 0 ) spectrum[pha]++;
    dmTableNextRow(event_table);
}

dmBlockClose(event_table);
dmDatasetClose(input_ds);
free(start);
free(stop);

output_ds =dmDatasetCreate("spectrum.fits");
out_table = dmBlockCreate(output_ds, "SPECTRUM", "TABLE");
channel = dmColumnCreate(out_table,"CHANNEL",dmLONG,0,"channel","Pulse height channel");
counts  = dmColumnCreate(out_table,"COUNTS",dmLONG,0,"count","Spectrum counts");
rate    = dmColumnCreate(out_table,"RATE",dmDOUBLE,0,"count/s","Count rate");
dmKeyWrite_d(out_table, "EXPOSURE", livetime, "s", "Livetime");
dmKeyWrite_l(out_table, "CHANMIN",  0, "channel", "Min PHA channel" );
dmKeyWrite_l(out_table, "CHANMAX",  MAXPHA, "channel", "Max PHA channel");
dmKeyWrite_c(out_table, "CHANTYPE", "PHA", " ", "PH binning type");
```

```
for (pha = 0; pha <= MAXPHA; pha++)
{
    dmSetScalar_l(channel, pha);
    dmSetScalar_l(counts, spectrum[pha]);
    dmSetScalar_d(rate, spectrum[pha]/livetime);
    dmTableNextRow(out_table);
}

dmDatasetClose(output_ds);
```

The same program can be written using explicit row data buffers and one-block-at-a-time dataset handling (some details omitted where the same as the previous version). The one-block-at-a-time dmDatasetTable and dmDatasetImage routines are a convenience for the special case of a dataset with one block in it, which is very common. It minimizes the number of handles floating around in the program.

```
dmDataset* dataset;
dmBlock* gti_table;
dmBlock* event_table;
dmBlock* out_table;
double livetime;
long spectrum[MAXPHA+1];
long pha;
double tzero;
struct { double start; double stop; } gti;
struct { long pha; short status; } event;
struct { long channel; long counts; double rate; } row;

dataset = dmDatasetTableOpen("bas.fits");
gti_table = dmTableOpenSelect(dataset,"STDGTI","START,STOP",NULL);
livetime = 0.0;

while(dmTableGetRow(gti_table,&gti) != dmNOMOREROWS)
      livetime += gti.stop - gti.start;

dmDatasetTableClose(gti_table);

#if FILTERING_IMPLEMENTED
event_table=dmDatasetTableOpen( "bas.fits[STDEVT][select PHA:i,STATUS:s]" );
#else
dataset = dmDatasetTableOpen("bas.fits");
event_table = dmTableOpenSelect(dataset,"STDEVT","PHA,STATUS",NULL);
#endif

dmKeyRead_d(event_table,"MJDREF",&tzero);
while(dmTableGetRow(event_table, &event) != dmNOMOREROWS)
      if (event.status == 0) spectrum[event.pha]++;

dmDatasetTableClose(event_table);
```

```
out_table = dmDatasetTableCreate("spectrum.fits[SPECTRUM]");
channel = dmColumnCreate(out_table,"CHANNEL",dmLONG,0,"channel","Pulse height channel");
counts  = dmColumnCreate(out_table,"COUNTS",dmLONG,0,"count","Spectrum counts" );
rate    = dmColumnCreate(out_table,"RATE",dmDOUBLE,0,"count/s","Count rate");

dmKeyWrite_d(out_table, "EXPOSURE",livetime, "s", "Livetime");
dmKeyWrite_l(out_table, "CHANMIN", 0, "channel", "Min PHA channel");
dmKeyWrite_l(out_table, "CHANMAX", MAXPHA, "channel", "Max PHA channel");
dmKeyWrite_c(out_table, "CHANTYPE","PHA", " ","PH binning type");

for (pha = 0; pha <= MAXPHA; pha++)
{
    row.pha = pha;
    row.counts = spectrum[pha];
    row.rate   = spectrum[pha]/livetime;
    dmTablePutRow( out_table, &row );
}

dmDatasetTableClose( out_table );
```

Better yet, once we have implemented the correct special cases, you should be able to do:

```
dmDataset* dataset;
dmBlock* event_table;
double livetime;
double* start;
double* stop;
long spectrum[MAXPHA+1];
long ngti;
struct { long pha; short status; } event;

dataset = dmDatasetOpen("bas.fits");
event_table=dmTableOpenSelect(dataset,"STDEVT","PHA,STATUS",NULL);
dmKeyRead_d( event_table, "MJDREF", &tzero );
dmSubspaceColGet_d( dmSubspaceColOpen( event_table, "TIME" ), &start,
       &stop, &ngti );
livetime = 0.0; for ( i= 0;i<ngti;i++ ) { livetime += stop[i] - start[i]; }

while(dmTableGetRow( event_table, &event ) != dmNOMOREROWS)
     if ( event.status == 0 ) spectrum[event.pha]++;
```

```
dmDatasetTableClose( event_table );
```

In this version, we don't ever see the GTI table explicitly. At the scientific level, GTI is just the filter on the time attribute, you don't care that in a FITS file it's stored in a separate extension. In fact, in a QPOE file the GTIs are not stored in a separate table. So it's important to provide this level of abstraction if you want the program to work on either QPOE or FITS files.

2 General DataModel Information

2.1 Online ASC DataModel References

This document is kept online at http://cfa-www.harvard.edu/ mnoble/ascdm. At the moment the DataModel is available only for internal AXAF Science Center use. The source will be made available via FTP and WWW at the time of a general release. Email the ASC DataModel alias (ascdm@cfa.harvard.edu), Michael Noble (mnoble@cfa.harvard.edu), or Jonathan McDowell (jcm@cfa.harvard.edu) for further information, or to obtain the source code prior to the general release.

2.2 Configuration and Sample Code

The doc directory in the DataModel distribution contains additional notes on installation, configuration, and use of the DataModel, as well as HTTP references to other astronomical software upon which the DataModel is layered.

The examples directory contains several programs and makefiles that can be used both as a test of the installation/configuration and as sample code.

In brief, the primary requirements for building an ASC DataModel program are:

```
- ensure #include "ascdm.h" appears in your source
- ensure your makefiles reference the Makevars.ascdm present in
  the root of the DataModel distribution tree
- ensure your compilation and link rules reference the appropriate
  DataModel macros specified within Makevars.ascdm.
```

2.3 Structure Objects

In support of the logical abstractions provided by the ASC DataModel, three object types are defined. Instances of these structures should be created and modified only through use of the access routines specified in this document. Direct access of the data structure internals may jeopardize the integrity of your application and data, and hence should be avoided.

- dmDataset*: pointer to an ASC dataset
- dmBlock*: pointer to an ASC datablock (ie, table or image)
- dmDescriptor*: pointer to an ASC data descriptor

2.4 Memory Management

To a large extent the ASC DataModel does not require the user to worry about details of memory management. For example, it is not necessary to free memory associated with individual dmDescriptor, dmBlock, or dmDataset pointers. Memory allocated to blocks and the descriptors they may contain will be freed when block is closed via an appropriate routine call. Similarly, memory allocated to dmDataset pointers will be freed when the dataset is properly closed.

Another example concerns routines that return character strings. Rather than have them return char* arrays, these routines instead write into a pre-allocated char* parameter, up to a specified maximum length parameter value (e.g., dmBlockGetName).

Despite these attempts, there are still instances when the DM user will need to explicitly deal with memory management. For example, array memory allocated by routines that return array pointers (e.g., dmTableOpenColumnList, dmBlockGetKeyList, or dmGetArrayDimensions) will need to be explicitly freed. In these cases, though, ONLY the array memory need be freed, not the individual array elements. Further details can be found in the accompanying function descriptions and code samples.

2.5 Defined Types

The DataModel defined types have integral values with symbolic names as listed.

2.5.1 dmDataType

Each dmDataType corresponds to either a C built-in type or DataModel typedef.

dmDataType	String	Meaning	Data/Class Type
dmSHORT	S	2 byte integer	dmshort, short
dmLONG	L	4 byte integer	dmlong, long
dmFLOAT	F	4 byte IEEE real	dmreal, float
dmDOUBLE	D	8 byte IEEE real	dmdouble, double
dmTEXT	C	String	dmString
dmBLOCKREF	BR	String, reference to block	typedef blockref dmString
dmBOOL	Q	Logical	typedef dmlogical int
dmBYTE	UB	1 byte unsigned	dmbyte, unsigned char
dmUSHORT	US	2 byte unsigned	dmushort, unsigned short
dmULONG	UI	4 byte unsigned	dmulong, unsigned long

The dmshort class is a machine-dependent #define to a 2 byte integer type; on many machines short will be equivalent. Same goes for other types. The logical and blockref types are typdef'd not #defined to make sure they are compiler-distinct from integer and string. Other types may be added later.

2.5.2 dmBlockType

The dmBlockType describes whether a block is a table, image, or something else.

dmBlockType	String
dmTABLE	TABLE
dmIMAGE	IMAGE
dmUNKNOWNBLOCK	UNKNOWN

2.5.3 dmDescriptorType

dmDescriptorType	String
dmCOLUMN	for columnar table data access
dmKEY	for header keyword access
dmIMAGEDATA	for access to image data
dmCOORD	for coordinate transform descriptors
dmSUBSPACE	for data subspace descriptors

Special cases: a special type of dmCOORD is an image axis; a special type of dmCOLUMN is a scalar column descriptor which is really one component of a vector column.

2.5.4 dmElementType

The DataModel considers tabular column and image data in terms of "cells," the intersection of a row and column, each of which contain one or more "elements," each of which in turn represent data in terms of the fundamental dmDataTypes. Cells can either be scalar, 1-dimensional arrays, or N-dimensional arrays, with constituent element types of:

dmElementType	String	Meaning
dmVALUE	V	Value (value)
dmRANGE	R	Range (min,max)
dmINTERVAL	I	Interval (value,min,max)

Note that elements can be multidimensional. For example, consider a cell containing elements representing points in Cartesian 3D space. Each (x,y,z) triple would be a dmValue element of dimensionality 3. Note that since cell dimensionality is independent of element dimensionality, it would still be possible to define the cell in question here as a scalar cell - meaning each cell would contain only 1 (x,y,z) triple.

2.5.5 dmKernelHint

The supported values for dmKernelHint are:

dmBINTABLE	hints to table creation routines that subsequent table constructions produce BINARY tables (default)
dmASCIITABLE	hints to table creation routines that the subsequent table constructions produce ASCII tables

2.6 Other Programming Considerations

2.6.1 Error Handling and Diagnostics

Most of the ASC DataModel routines indicate completion status either by returning an integer status code or by returning unusual values (e.g., NULL pointers or negative row numbers). Regardless of whether or not a DataModel API function provides explicit error state indication, the call completion status can be determined by using the **dmGetError** and **dmGetErrorMessage** routines.

The **dmFAILURE** status code indicates some error condition exists, while **dmSUCCESS** indicates the call completed successfully. Other status codes and return values are listed as appropriate with the associated APi functions.

Use **dmGetVersion** to find the current DataModel release version. This may be needed if you send email to the ASC about possible bugs. Also, for those familiar with configuration management issues, the "what" utility can be used to scan the DataModel object code library to provide more detailed code module version information.

Use **dmDatasetPrintKernel** and **dmBlockPrintKernel** to inspect the contents of the file at the kernel level, bypassing the layer of interpretation imposed by the DataModel. Use **dmBlockGetNoKernelKeys** and **dmBlockGetKernelKey** to inspect header entries at the kernel level.

2.6.2 Counting in the DataModel

The ASC DataModel uses a ones-based counting system. That is, the smallest block number, key number, image pixel coordinate, column number, or row number will always be 1.

2.6.3 Initialization Routines

It is not necessary within DataModel programs to explicitly call the IRAF initialization routine(s) when linking against the IRAF/QPOE kernel, as the necessary IRAF initialization(s) will be performed internally by the DataModel. In fact, since one of the goals of the DataModel is to free the user from file-format specifics, the use of any explicit file-format specific functionality is discouraged.

2.6.4 Multithreading

At the time of this writing the ASC DataModel is NOT thread-safe. The decision to implement the DataModel in this manner was primarily due to the fact that the DataModel library sits atop numerous other astronomical software libraries, most of which are themselves NOT thread-safe.

3 Introduction to the ASC dm library routines

Although there are a large number of routines in the library, they can be grouped fairly simply. Many routines are used in multiple contexts; for instance, the same routine may be used to read data from a table or an image. This section organizes the routines by usage, briefly describing the routines to use in each context - for instance, table routines are grouped together in one subsection, and the same routine may be referred to in several subsections. However, the details of the routine are not given here, but in the final section of the document where all the routines are described in alphabetical order.

3.1 Dataset operations

We first describe operations at the dataset level. Recall that each dataset contains a series of table blocks and/or image blocks.

3.2 Opening and closing files

To open an existing dataset, use either the dmDatasetOpen routine or (if you are only interested in one table or image in the dataset) the dmTableOpen/dmImageOpen routines. The former returns a pointer of type dmDataset*, and you can then use that pointer to open various blocks (tables or images) in the dataset using dmBlockOpen. The latter directly returns a pointer of type dmBlock*. Each of the Open routines has a corresponding Close routine and a corresponding Create routine for opening a new object to write to.

Table 1: Open/close routines

dmDatasetOpen	Open dataset by name, return dmDataset*
dmDatasetCreate	Same for creation
dmDatasetClose	Close a dataset.
dmBlockOpen	Open block (table/image) in dataset, return dmBlock*
dmBlockCreate	Same for creation, don't use for images
dmDatasetCreateTable	Block create for table
dmDatasetCreateImage	Block create for image, specify size
dmBlockClose	Close block opened by dmBlockOpen/Create
dmTableOpen	Open table and dataset at same time, return dmBlock*
dmTableCreate	Same for creation
dmTableClose	Close dataset opened by dmDatasetTableOpen/Create
dmImageOpen	Open image and dataset at same time, return dmBlock*
dmImageCreate	Same for creation
dmImageClose	Close image

If you open a dataset using the dmTable or dmImage routines, you only have a block pointer. If you then need the dataset pointer you can get it with the **dmBlockGetDataset** routine.

Another routine to create blocks is the **dmBlockCreateCopy** routine, which copies the structure of an existing block without its data.

3.2.1 Navigating within a dataset

Most of the time we work with a single block within the dataset. How do you get to the block you want? There are several ways. You may access the blocks sequentially, or by number, or by name.

To access the blocks sequentially, use the **dmDatasetNextBlock** routine. This routine will open the next block, (the first time, it will open the first block in the dataset), and repeated calls will go through all the blocks until the end of the dataset, when it will return null. The

dmDatasetGetCurrentBlockNo inquiry routine returns the number of the most recent block to have been opened. The **dmDatasetAdvanceBlocks** routine moves ahead or back by a specified number of blocks, so dmDatasetNextBlock is equivalent to calling dmDatasetAdvanceBlocks with an argument of 1.

To access the blocks by number, use **dmDatasetMoveToBlock**.

The **dmDatasetGetNoBlocks** inquiry routine returns the total number of blocks in the file.

The **dmDatasetGetBlockType** routine is used to find out whether the block you are about to open is a TABLE or an IMAGE. If you have opened the block (i.e. you have a dmBlock* pointer) you can use the **dmBlockGetType** or **dmBlockGetTypeStr** routines to find out what kind of block you have.

3.2.2 Kernel related routines

The following routines allow control over which kernel is used to make new files. In normal use we want the existence of different kernels to be invisible to the programmer, so we separate these calls out instead of making the kernel id an argument to CreateDataset as in the EDS layer.

Use **dmKernelSetCreate** to specify the kernel to be used when creating new datasets from scratch. Use **dmKernelSetCopy** to specify the kernel to be used when making a copy from an existing dataset (usually its kernel is copied too).

Use **dmKernelGetCreate** and **dmKernelGetCopy** to inspect the current settings.

dmKernelGetList tells you what kernels are available at run-time.

dmDatasetGetKernel is used to find out which ETOOLS kernel (i.e. which underlying disk format) is being used for a particular dataset.

3.2.3 Auxiliary dataset routines

There are some auxiliary routines which are used less often to manipulate datasets.

- **dmDatasetCopy** is used to make a copy of an entire dataset.
- **dmDatasetDelete** deletes a dataset on disk.
- **dmDatasetRename** renames a dataset.
- **dmDatasetAccess** is used to test whether a dataset exists, prior to opening it.
- **dmDatasetPrint** is used as a diagnostic routine to examine the structure and contents of a dataset.

3.3 Tables

3.3.1 Opening a table

You can open an existing table in the following ways:

- Open the next block in a dataset with **dmDatasetNextBlock**.
- Open a numbered block in a dataset with **dmDatasetMoveToBlock**.
- Open a block by name with **dmBlockOpen**.
- Open a block and a dataset at the same time using **dmTableOpen**.
- Open a table for row-based I/O using **dmTableOpenSelect** (see Row Based I/O below).

In each of these cases except for dmTableOpen you must check that it is a table and not an image, using **dmDatasetGetBlockType** or **dmBlockGetType**, and call **dmBlockClose** when you are done with the block. For **dmTableOpen** you are guaranteed that it is a table, and you must call **dmTableClose** when you are done, which releases both the block and the parent dataset at the same time.

You can delete the table entirely by using the **dmBlockDelete** call.

You can create a new table and dataset using **dmTableCreate**. To create a dataset with multiple tables and/or images, use **dmDatasetCreate** to make the dataset and **dmDatasetCreateTable,dmDatasetCreateImage** to create new tables and/or images. For fine control, use **dmSetKernelOption** first, to control details of the disk format used (e.g. FITS ASCII tables versus the default BINTABLE).

3.3.2 Basic table properties

.

- **dmBlockGetName** returns the name of the table.
- **dmBlockGetDataset** returns a pointer to the dataset of which the table is a member.
- **dmBlockGetNo** returns the number of the block in the dataset.
- **dmTableGetNoCols** returns the number of columns in the table.

3.3.3 Creating table structure

The **dmColumnCreate** call creates a scalar column with a specified data type and name. Repeated calls may be used to create all the columns for a new table. After you start writing data to the table, you can't add any more columns.

To make an array column (a 1-dimensional array of elements in each table cell), use the **dmColumnCreateArray** or **dmColumnCreateVarArray** calls.

You can also store an entire n-dimensional array in each table cell, using a column created with the **dmColumnCreateNDArray** routine.

The data model introduces the concept of vector columns, in which several columns are grouped together each with their own name but also with a common name. To create such a vector column, use the **dmColumnCreateVector** call.

Another kind of grouped column imposes a particular meaning on the grouping, using the concept of compound element types (also known as Intervals). This allows us to store ranges of values rather than point values. For instance, the Good Time Intervals (START,STOP) are more elegantly handled as a RANGE element for the TIME variable. Columns of this kind are created with the **dmColumnCreateElement** routine.

Combining the high level constructs to produce arrays of vectored compound element types may be done using the **dmColumnCreateGeneric** routine; all the other column create routines are special cases of this. You can define a set of columns at once with **dmTableCreateColumns** (scalar columns only) or **dmTableCreateGenericColumns** (generic columns).

The **dmColumnInsertNo** and **dmColumnInsertAfter** routines allow you to create a column out of order, useful when copying a block.

3.3.4 Navigating in the table

To start with, you are always at the first row of the table. Repeated read/write operations will not change the row. To move to another row, use **dmTableNextRow** or **dmTableSetRow**. To find out which row you are at, use **dmTableGetRowNo**. The routine **dmTableGetNoRows** tells you how many rows are in the table.

To read or write data to the table, you can use cell-based I/O which operates on one column of one row at a time, column-based I/O which reads/writes multiple rows of a single column, or row-based I/O which operates on a whole row at once using a C structure.

3.3.5 Cell-based I/O: introduction

To use cell-based I/O you must first obtain descriptors for each column you wish to access, using **dmTableOpenColumn** or **dmTableOpenColumnNo**. You can get the entire list of columns using **dmTableOpenColumnList**. Then you can navigate the rows using **dmTableNextRow** or **dmTableSetRow**, and use the GetScalar or SetScalar calls and their relatives to read or write the data.

3.3.6 Column properties

To get or alter the properties of a column, use the generic descriptor dmGet/dmSet calls:

- **dmGetName**, **dmSetName** - get/set name of column
- **dmGetUnit**, **dmSetUnit** - get/set unit of column
- **dmGetDataType** - get data type of column (cannot be changed)
- **dmGetDesc**, **dmSetDesc** - get/set descriptive comment for column
- **dmGetArrayDim** - get array dimensionality for column (cannot be changed)
- **dmGetElementDim** - get vector dimension for column (cannot be changed)
- **dmGetElementType** - get element type of column (cannot be changed)
- **dmGetDisp**, **dmSetDisp** - get display format hint for column
- **dmColumnGetNo** - get number of column in table

If the column has nonzero array dimensionality, the **dmGetArrayDimensions** and **dmGetArraySize** routines may be used to find the shape of the array and the total number of array elements per cell.

If the column is a vector column, the **dmGetCptName**, **dmSetCptName** routines can be used to find or alter the name of each vector component and **dmGetElementDim** can be used to find the number of components. For example, one might have a descriptor whose name is DETPOS, with 2 components DETX and DETY representing different axes. This is in contrast to an array descriptor which might be say DETX(2), with 2 values from the same axis. One may even have vectored array descriptors but this is not encouraged. The routine **dmGetCpt** returns a scalar column descriptor corresponding to a single component of a vector column.

Each descriptor also has an element type and, possibly, an interval type. The element types supported at release R1/R2 are dmVALUE, dmRANGE, and dmINTERVAL. The dmRANGE and dmINTERVAL element types are understood to describe closed intervals. Descriptors also have an Interval Type which allows you to specify open or semi-open intervals, but this will not be implemented (**dmGetIntervalType**) until at least R3.

To get all the information for a descriptor in a single call, use the **dmDescriptorInfo** call.

Use **dmDescriptorDelete** to delete a column.

To check that your descriptor really is a column and not a key, you can use the **dmDescriptorGetType** routine.

3.3.7 Cell-based I/O read and write

To read/write a scalar cell value from/to the current row, use the column descriptor and the **dmGetScalar/dmSetScalar** call.

Like FITSIO, our default approach to getting data in and out of tables is cell-based I/O, where we work on one row and column at a time. Thus, the dmTableOpenColumn routine returns a dmDescriptor* for the column:

```
dmDescriptor* pha_col = dmTableOpenColumn( table, "PHA" );
```

Reading from this column gets the value from the current row, which initially is the first row of the table:

```
pha = dmGetScalar_l( pha_col );
```

This dmGetScalar routine gets a single value from a scalar type column (the usual sort). dmGetScalar has various versions subscripted with the data type of the quantity to be returned; thus dmGetScalar_l returns a value that can be stored as a 4 byte integer. To get the value for the next row, we must advance the current row:

```
dmTableNextRow( table );
```

3.3.8 Cell-based I/O: complicated cases

As well as scalar columns, our data can be vector columns, 1-D array columns, N-D array columns, and vector array columns. The dmGetScalar and dmPutScalar routines each have cousins to handle these more complicated types of data. For instance, the dmGetArray family returns a 1-D array of values for an array column.

To handle array and vector columns, use **dmGetArray/dmSetArray**, **dmGetVector/dmSetVector**.

To handle scalar columns with compound element types (ranges and intervals), use **dmGetElement/dmSetElement**. **dmSetLimit** may be used to denote an upper limit, and **dmDescriptorIsUpperLimit** may be used to test for one.

For array columns, to read or write a rectangular sub-array, use **dmImageDataGetSubArray**, **dmImageDataSetSubArray**. To read or write a single pixel, use **dmImageDataGetPixel**, **dmImageDataSetPixel**.

3.3.9 Column-based I/O

There is another family of routines, **dmGetScalars/dmSetScalars**, which reads/writes many rows at once. It may be used for column-based I/O, which is efficient if the table is small or if the whole table has been read into memory. However, full column-based I/O is not efficient when working on a large FITS file which CFITSIO has buffered to be only partly in memory, since the whole table must be reread each time you read in a column. In this case, you may use dmGetScalars with an intermediate number of rows, and read a batch of records at a time.

A companion set of routines, **dmGetVectors/dmSetVectors**, supports vector columns.

3.3.10 Row-based I/O

The most convenient way to access data in a table when you know what data you want is to use the row-struct I/O method. In this method, you define a C struct containing the information you are interested in for each row of the table. For example, suppose that you know the table contains the columns PHA, STATUS and TIME of types long, short and double respectively. Then define the struct:

```
struct { long pha; short status; double time } myrow;
```

If this structure corresponds exactly to the table structure, you can directly use the dmTableGetRow routines to fill `myrow` with the data and use e.g. myrow.status as a variable. If the table might have extra rows or have the rows in a different order, you have to tell the dm library explicity what your myrow struct contains. To do this, use the **dmTableOpenSelect** routine.

The **dmTableGetRow** routine returns the data for the current row in the row-structure. The **dmTablePutRow** routine writes the row-structure to the internal row buffers and hence to the table. Both of these routines advance the row pointer, so you do NOT need to call **dmTableNextRow** when using these routines.

The **dmTableCopyRow** routine copies a row from one table to another, when the second table was created from the first by **dmBlockCreateCopy**.

The advantages of row-based I/O are offset by a lack of type checking on the data. Also, if you don't know what data is in the table in advance (generic table browsing or calculation tools), you have to use cell-based I/O since you can't declare a row struct at run time.

3.3.11 Preferred Axes

Typically a table may contain a small number of crucial columns and a larger number of columns with 'extra' information. The user will often regard the table as being either a tabulation of one dependent variable Y against independent variables X1, X2, ...XN ('histogram interpretation'), in other words a function Y(X1,X2,...XN); in this case usually the values of (X1,..XN) do not repeat. Alternatively, the table may be a list of measurements of independent variables X1, ... XN, which the user may want to correlate one with another ('raw table interpretation') or make a histogram of as N(X1,X2,...XN) ('event list interpretation'). In 'first look' type software, it is useful to be able to figure out which columns of the table correspond to Y, X1, ..XN and which are 'extra' information. The answer to this for a given table may depend on what the user is interested in, but often there are suitable defaults. For example, a photon event list might reasonably default to some particular pair of spatial coordinates (X,Y), and a spectrum histogram might default to counts as a function of channel: COUNTS(CHANNEL). We provide a convention to record this information in the header of the table.

The **dmBlockSetPreferred** may be used to record the defaults in the table; lower level routines **dmBlockSetPrefAxes** and **dmBlockSetPrefWeight** are also supplied to set the independent and dependent variables separately using descriptors. The **dmBlockGetPrefAxes** and **dmBlock-GetPrefWeight** routines may then be used to extract the information.

3.4 Coordinate Descriptors

3.4.1 Coordinates

Columns in a table or the axes of an image may have coordinate systems attached to them. The coordinate system can be thought of as a 'virtual column' which is defined in terms of the original column. You get its dmDescriptor* using the dmDescriptorGetCoord routine. In the simple case of a scalar column with a linear coordinate transform, you get the standard transformation parameters CRPIX, CRVAL and CDELT using the dmCoordGetLinTransform routine.

To write a coordinate system on a table column or an image axis group, use the **dmCoordCreate** routines or the **dmCoordCreateInteger** routines. To make an image axis group (a 'physical coordinate system' in IRAF terminology), use **dmArrayCreateAxisGroup**.

A 'lookup transform' which uses an external table instead of a functional transform can be defined with the **dmCoordCreateLookup** routine.

To find the default coordinate associated with a descriptor (if any), call **dmDescriptorGet-Coord**. There may be more than one coordinate associated with a descriptor; **dmDescriptor-GetNoCoords** and **dmDescriptorGetCoordNo** may be used to get them all. Conversely, **dm-CoordGetParent** may be used to find the parent descriptor of a coordinate descriptor.

You can find the transform type using **dmCoordGetTransformType**, and the transform values CRPIX, CRVAL, CDELT using **dmCoordGetTransform**. To get the transform parameters, use **dmCoordGetParams**.

3.4.2 Coord values

Suppose you have a scalar dmDOUBLE column called TIME (descriptor `time` with a coordinate called DATE (descriptor `date = dmGetCoord( time )`). The value of TIME in the current row might be 14823.3 seconds; the corresponding value of DATE might be JD 2450423.52 days. To read the value of TIME, you use **dmGetScalar_d** on the column data descriptor `time`. To get the value of DATE for this row, you simply use **dmGetScalar_d** on the coordinate descriptor `date` instead.

However, if you want to find the DATE for some value of TIME which is not in the table, you must apply the transform explicitly by using **dmCoordCalc**. The inverse transformation is also provided, **dmCoordInvert**.

Example:

```
dmDescriptor* time = dmTableColumnOpen( table, "TIME" );
dmDescriptor* date = dmGetCoord( time );
dmTableNextRow( table );
double date_value = dmGetScalar_d( date );
double time_value = dmGetScalar_d( time );
double time_value2 = 45.8;
double date_value2;
double date_value3 = 45382.4;
double time_value3;
dmCoordCalc_d( date, &time_value2, &date_value2 );
dmCoordInvert_d( date, &date_value3, &time_value3 );
```

3.4.3 Coord properties

To get or alter the properties of a coord descriptor, use the generic descriptor dmGet/dmSet calls:

- **dmGetName**, **dmSetName** - get/set name of coord
- **dmGetUnit**, **dmSetUnit** - get/set unit of coord
- **dmGetDataType** - get data type of coord (cannot be changed)
- **dmGetDesc**, **dmSetDesc** - get/set descriptive comment for coord
- **dmGetArrayDim** - get array dimensionality for coord (always 0)
- **dmGetElementDim** - get vector dimension for coord (cannot be changed)
- **dmGetElementType** - get element type of coord (cannot be changed)
- **dmGetDisp**, **dmSetDisp** - get display format hint for coord

If the coord is a vector coord, the **dmGetCptName**, **dmSetCptName** routines can be used to find or alter the name of each vector component and **dmGetElementDim** can be used to find the number of components. The coord must have the same element dimension as its parent descriptor.

To get all the information for a descriptor in a single call, use the **dmDescriptorInfo** call.

To delete a coord, use the **dmDescriptorDelete** call.

3.5 Header keys

3.5.1 Header keys

Header keys are treated as table columns with a single row; they are present in both tables and images. You can create a new header key as follows:

- Use **dmKeyCreate** to create a descriptor for the key, and then use **dmSetScalar** to set its value.
- Use **dmKeyWrite** to create the descriptor and write the value, unit and description at the same time. This is usually the most convenient.
- Use **dmBlockMoveToKey**, **dmBlockMoveToKeyNo**, and **dmBlockAdvanceKeys** to reposition yourself in the header so that you can write keys out of order.

In later releases we will support array, compound element, and vector header keys. These may be written analogously:

- Use **dmKeyCreateGeneric** to create a descriptor for a generic key, and use various dmSet routines to set the values;
- or use **dmKeyWriteVector**, **dmKeyWriteArray**, **dmKeyWriteInterval** to write the values at the same time as creating the descriptor.

To find the total number of keys in the block, use **dmBlockGetNoKeys**.
To read a header key from a block, you have the following choices:

- Use **dmKeyOpen** to search for the key by name and return a descriptor for it.
- Use **dmBlockGetKey** to return a descriptor for a key given its number (order) in the header. Keys are numbered starting at 1. To get all the keys in the block, use **dmBlockGetKeyList**.
- Use **dmKeyRead** to search for the key by name, and return both a descriptor and the key's value, forced to a particular data type. If no key of that name is present, dmKeyRead returns a null descriptor (and zero or blank in the value). Use **dmKeyReadVector** to read vectored or array keys.
- To read or write a scalar key value when you already have its descriptor, use the **dmGetScalar/dmSetScalar** calls. You can use the **dmGetArray/dmSetArray**, **dmGetVector/dmSetVector**, **dmGetInterval/dmSetInterval** for more complicated kinds of key.

3.5.2 Key properties

To get or alter the properties of a key, use the generic descriptor dmGet/dmSet calls:

- **dmGetName**, **dmSetName** - get/set name of key
- **dmGetUnit**, **dmSetUnit** - get/set unit of key
- **dmGetDataType** - get data type of key (cannot be changed)

- **dmGetDesc**, **dmSetDesc** - get/set descriptive comment for key
- **dmGetArrayDim** - get array dimensionality for key (cannot be changed)
- **dmGetElementDim** - get vector dimension for key (cannot be changed)
- **dmGetElementType** - get element type of key (cannot be changed)
- **dmGetDisp**, **dmSetDisp** - get display format hint for key
- **dmKeyGetNo** gets the number of the key in the header.

If the key has nonzero array dimensionality, the **dmGetArrayDimensions** and **dmGetArraySize** routines may be used to find the shape of the array and the total number of array elements per cell.

If the key is a vector key, the **dmGetCptName**, **dmSetCptName** routines can be used to find or alter the name of each vector component and **dmGetElementDim** can be used to find the number of components. For example, one might have a descriptor whose name is DETPOS, with 2 components DETX and DETY representing different axes. This is in contrast to an array descriptor which might be say DETX(2), with 2 values from the same axis. One may even have vectored array descriptors but this is not supported for keys.

Each descriptor also has an element type and, possibly, an interval type. The element types supported at release R1/R2 are dmVALUE, dmRANGE, and dmINTERVAL. The dmRANGE and dmINTERVAL element types are understood to describe closed intervals. Descriptors also have an Interval Type which allows you to specify open or semi-open intervals, but this will not be implemented (**dmGetIntervalType**) until at least R3.

To get all the information for a descriptor in a single call, use the **dmDescriptorInfo** call.

To delete a key, use the **dmDescriptorDelete** call.

3.5.3 Comments

FITS-style COMMENT and HISTORY header information is supported via the **dmBlockWriteComment** and **dmBlockReadComment** routines.

3.6 Images

3.6.1 Opening an image

You can open an existing image in the following ways:

- Open the next block in a dataset with **dmDatasetNextBlock**
- Open a numbered block in a dataset with **dmDatasetMoveToBlock**.

- Open a block by name with **dmBlockOpen**
- Open a block and a dataset at the same time using **dmImageOpen**

In each of these cases except for **dmImageOpen** you must check that it is an image and not a table, using **dmDatasetGetBlockType**, and call **dmBlockClose** when you are done with the block. For **dmImageOpen** you are guaranteed that it is an image, and you must call **dmImageClose** when you are done, which releases both the block and the parent dataset at the same time. Once you have opened the image, if you want to access the image data or axis info (rather than just the header info) you have to get the descriptor for that image data using **dmImageGetDataDescriptor**.

You can delete the image entirely by using the **dmBlockDelete** call.

To create an image, you first create the image dataset or block using **dmImageCreate** or (if the dataset exists) **dmDatasetCreateImage**. You may then name the axes using **dmArrayCreateAxisGroup** or **dmArrayCreateAxisGroups**. **dmImageGetDataDescriptor** returns the newly created image data descriptor on which you can use **dmSetArray** or **dmImageDataSetPixel** to write the values.

3.6.2 Basic image properties

.

Images have a set of n axes (often n=2) each of which has a dimension (the length of the axis). They also have a set of pixel values arranged in an n-dimensional array. **dmImageGetDataDescriptor** returns a descriptor for the image data. You can then use **dmGetArray** on this descriptor to get the array of values, just as if the image was a cell in a table. Alternatively, you can use **dmImageDataGetPixel** to get the values one pixel at a time. Use **dmGetDataType** on the image data descriptor to find the data type of the pixel values. To find the dimensionality of the image, the **dmGetArrayDimensions** routine tells you what and how long each axis is.

- **dmBlockGetName** returns the name of the image.
- **dmBlockGetDataset** returns a pointer to the dataset of which the image is a member.
- **dmBlockGetNo** returns the number of the block in the dataset.

Example:

```
long* axes;
dmBlock* image = dmImageOpen("image.dat");
char name[MAXLEN];
dmBlockGetName(image,name,MAXLEN);
dmDescriptor* data = dmImageGetDataDescriptor(image);
```

```
dmDataType type = dmGetDataType(data);
naxes = dmGetArrayDimensions(data, &axes);
free(axes);
dmImageClose(image);
```

3.6.3 Image axes

- **dmArrayCreateAxisGroup** creates a descriptor for an axis. It names the axis and creates a unit coordinate transform from the pixel values to the descriptor.
- **dmArrayGetNoAxisGroups** returns the number of axis groups on the image.
- **dmArrayGetAxisGroup** returns descriptor for nth axis group.
- To find the physical coordinates at a particular pixel number in the image, use **dmCoordCalc** with the axis group as argument.
- To find the pixel value corresponding to particular physical coordinates, use **dmCoordInvert** with the axis group as argument.
- To find the world coordinates for the image, use **dmGetCoord** on the axis group. This returns the physical to world transformation.

Example:

```
long logical[2] = { 20, 20 };
double physical[2];
double world[2];
dmBlock* image = dmImageOpen( "myimage.fits[1:100,500:600]" );
dmDescriptor* imageData = dmImageGetDataDescriptor( image );
long* axes;
long naxes = dmGetArrayDimensions( imageData, &axes );
long ngroups = dmArrayGetNoAxisGroups( imageData );
dmDescriptor* group1 = dmArrayGetAxisGroup( imageData, 1 );
long dim = dmGetElementDim( group1 );
dmDescriptor* world_wcs = dmGetCoord( group1 );
dmCoordCalc_l( group1, logical, physical );
dmCoordCalc_d( world_wcs, physical, world );
free( axes );
```

3.6.4 Image data

- **dmImageGetDataDescriptor** returns the image data descriptor.
- To read the data from the array, use the **dmGetArray** call.
- To write the data to the array, use **dmSetArray**.
- To read or write a rectangular sub-array, use **dmImageDataGetSubArray**, **dmImageDataSetSubArray**.
- To read or write a single pixel, use **dmImageDataGetPixel**, **dmImageDataSetPixel**.
- To interpolate in the image, use **dmImageDataInterpolate**.

3.6.5 Image properties

To get or alter the properties of a Image, use the generic descriptor dmGet/dmSet calls on the image data descriptor.

- **dmGetName**, **dmSetName** - get/set name of Image data quantity
- **dmGetUnit**, **dmSetUnit** - get/set unit of Image pixel values
- **dmGetDataType** - get data type of Image (cannot be changed)
- **dmGetDesc**, **dmSetDesc** - get/set descriptive comment for Image
- **dmGetArrayDim** - get array dimensionality for Image (cannot be changed)
- **dmGetArrayDimensions** - get shape of array (size of each axis)
- **dmGetArraySize** - get total number of array elements per cell.
- **dmGetElementDim** - get vector dimension for Image pixels (cannot be changed, usually 1)
- **dmGetElementType** - get element type of Image pixels (cannot be changed, usually dmVALUE)
- **dmGetDisp**, **dmSetDisp** - get display format hint for Image pixel values

If the Image is a vector Image (not supported until R3+), the **dmGetCptName**, **dmSetCptName** routines can be used to find or alter the name of each vector component and **dmGetElementDim** can be used to find the number of components. Each descriptor also has an element type and, possibly, an interval type. The element types supported at release R1/R2 are dmVALUE, dmRANGE, and dmINTERVAL. However, images almost always have an element type of dmVALUE.

To get all the information for a descriptor in a single call, use the **dmDescriptorInfo** call.

3.6.6 Image pixel lists

An alternate way of representing an image is as a list of pixels and their values. This is convenient for sparse arrays, and is related to the event list representation. The **dmImageDataGetPixlistSize** routine returns the number of nonzero pixels in the cell. **dmImageDataGetPixlist** and **dmImageDataSetPixlist** are used to read and write image data in the form of pixel lists. Note that in these routines the pixel lists are the interface to the data, but the actual storage of the data in the file is still the standard image format (whatever that is for the kernel in question).

3.7 Data Subspace

3.7.1 Subspace columns

We want to record in the file a description of how the data has been filtered. Although in the underlying file format this may be implemented using header keywords, we treat this information specially at the data model level.

To store a filter, use the dmSubspaceColCreate routines. For example, in our earlier sample code we wrote two header keys descriping the PHA range:

```
dmKeyWrite_l( out_table, "CHANMIN",  0, "channel", "Min PHA channel" );
dmKeyWrite_l( out_table, "CHANMAX",  MAXPHA, "channel", "Max PHA channel" );
```

We might instead write

```
phamin = 0;
phamax = MAXPHA;
dmSubspaceColCreate_l( out_table, "PHA", "channel", &phamin, &phamax, 1 );
```

The difference is that the file now intrinsically knows that 0 and MAXPHA are the min and max values that descripe the PHA variable. Similarly we might write

```
dmSubspaceColCreate_d( out_table, "TIME", "s", start, stop, ngti );
```

The SubspaceColCreate code will recognize TIME as a special case and store the array of values in a separate GTI table.

To store a new filter, use **dmSubspaceColCreate** or **dmSubspaceCreateRegion**. To later alter its values, use **dmSubspaceColSet** to overwrite old values or **dmSubspaceColUpdate** to intersect new values with old values.

To find an existing filter, use **dmSubspaceColOpen** and then read its values using **dmSubspaceColGet**. These routines may be combined as **dmSubspaceColRead**. For a region filter, use **dmSubspaceColOpen** followed by **dmSubspaceGetRegion**.

3.7.2 Subspace column properties

To get or alter the properties of a subspace column descriptor, use the generic descriptor dmGet/dmSet calls:

- **dmGetName**, **dmSetName** - get/set name of subspace descriptor
- **dmGetUnit**, **dmSetUnit** - get/set unit of subspace descriptor
- **dmGetDataType** - get data type of subspace descriptor (cannot be changed)
- **dmGetDesc**, **dmSetDesc** - get/set descriptive comment for subspace descriptor
- **dmGetArrayDim** - get array dimensionality for subspace descriptor (cannot be changed)
- **dmGetElementDim** - get vector dimension for subspace descriptor (cannot be changed)
- **dmGetElementType** - get element type of subspace descriptor (cannot be changed)
- **dmGetDisp**, **dmSetDisp** - get display format hint for subspace descriptor

The subspace descriptor usually has array dimensionality 1; **dmGetArraySize** routine may be used to find the shape of the array and the total number of array elements per cell. The **dmSubspaceColSet** routines are special in that they can change the number of array elements for the subspace.

If the subspace descriptor is a vector subspace descriptor, the **dmGetCptName**, **dmSetCptName** routines can be used to find or alter the name of each vector component and **dmGetElementDim** can be used to find the number of components. For example, one might have a descriptor whose name is DETPOS, with 2 components DETX and DETY representing different axes. This is in contrast to an array descriptor which might be say DETX(2), with 2 values from the same axis. One may even have vectored array descriptors but this is not encouraged.

Each descriptor also has an element type and, possibly, an interval type. The element types supported at release R1/R2 are dmVALUE, dmRANGE, and dmINTERVAL. The dmRANGE and dmINTERVAL element types are understood to describe closed intervals. Descriptors also have an Interval Type which allows you to specify open or semi-open intervals, but this will not be implemented (**dmGetIntervalType**) until at least R3.

To get all the information for a descriptor in a single call, use the **dmDescriptorInfo** call.

To delete a filter descriptor, use the **dmDescriptorDelete** call.

3.7.3 Accessing subspace columns

- **dmBlockGetNoSubspaceCols** returns the total number of filters.
- **dmBlockGetNoSubspaceCpts** returns the number of separate components in the subspace (see the abstract design document for details).
- **dmBlockGetSubspaceColNo** gets a filter by number.
- **dmBlockGetSubspace** returns the full list of descriptors for the filters in the subspace.
- **dmBlockSetSubspaceCpt** sets the value of the subspace component number, used by dmSubspaceColCreate etc.
- **dmBlockGetCurrSubspaceCpt** returns the current subspace component, used by dmSubspaceColUpdate, etc.

3.7.4 Subspace routines

These routines may actually parse the data subspace to apply filtering constraints.

- **dmBlockInSubspace** tests whether a set of values is in the data subspace or not.
- **dmBlockIntersectSubspace** creates a new data subspace which is the intersection of two others.
- **dmFilterIntersect** sets the value of one filter descriptor to be the intersection of two others.
- **dmBlockPrintSubspace** is a diagnostic routine to show the current values in the data subspace.

4 Alphabetical Routine Descriptions

The remainder of this document focuses on the functions provided by the ASC DataModel APi. Some routine descriptions include multiple ‘contexts’ indicating behaviour which may need to be tested separately and which may be implemented in different releases. Each context is flagged with the DataModel release number it is expected to be supported in, e.g. (R2) for release 2.

4.1 dmArray Routines

4.1.1 dmArrayCreateAxisGroup

dmDescriptor* dmArrayCreateAxisGroup(dmDescriptor* dd, char* name, dmDataType type, char* unit, char** cptnames, integer dim)

(R1.7) This routine is used to add names to the axes of an image, and initialize a 'logical to physical' coordinate transform in which the actual software binned pixels of the image may be related to an original, usually finer, pixel intended to be used as the physical detector pixel. The actual pixels of the image are called the 'logical coordinates', and the original unbinned pixels are called the 'physical coordinates'.

The axes of an n-dimensional array are collected together into 'axis groups' (see the abstract design). Some images have axes which are unrelated to one another; e.g. a TIME versus ENERGY versus VELOCITY 3-d image would have three axis groups each of dimension one. Other images have axes which are paired, e.g. a sky X vs Y 2-d image would have one axis group of dimension 2. The distinction is that a later world coordinate transformation mapping X,Y to the celestial sphere needs to be anchored to both axes at once, the X transform is not independent of the Y transform. While there are other examples, a good initial approach is to assume that axes representing positional information are grouped together, while other axes are independent.

The routine creates an axis group of dimension dim on the next available set of axes. It returns NULL and sets an error if we run out of axes; clearly, the sum of the 'dim' values on all calls to this routine for a given array descriptor should equal the number of axes on the array. Creates and returns a dmCOORD descriptor whose parent is the just created axis group, with the given name and component names and element dimensionality dim. Initializes the transform to the unit transform. The transform can then be reset with dmCoordSetTransform_l. This version works on descriptors of type Image Data.

(R3.0) Support for table cell images.

dmDescriptor* dmArrayCreateAxisGroups(dmDescriptor* dd, char** name, dmDataType* type, char** unit, char** cptnames, long* dim, long ngroups)

(R1.7) Like the above, but lets you define multiple groups at once. The array cptnames is of size n (number of axes), the other arrays are of size ngroups.

4.1.2 dmArrayGetAxisGroup: Get image axis descriptors

dmDescriptor* dmArrayGetAxisGroup(dmDescriptor* data, long axisGroupNo)

The argument for this routine is the image data descriptor, while a descriptor for the nth axis group of the image is returned.

Context 1 (R1.7): Return the DD for the given axis group number; this DD can be used to find the unit, name, etc. of the axis group.

Context 2 (R3): Support for table cell images.

4.1.3 dmArrayGetNoAxisGroups

long dmArrayGetNoAxisGroups(dmDescriptor* imageData)

Context 1 (R1.7): Return number of axis groups.
Context 2 (R3.0): Support for table cell images.

4.2 dmBlock Routines

4.2.1 dmBlockAdvanceKeys

dmDescriptor *dmBlockAdvanceKeys(dmBlock* block, long no)

(R1.5): Move in header by a relative number of keys, which may be negative. Returns NULL if the resulting position is off the end of the header in either direction, or a pointer to the resulting descriptor otherwise.

4.2.2 dmBlockClose

int dmBlockClose(dmBlock* block)

(R1): Close a datablock within the dataset, but keep the dataset open.

4.2.3 dmBlockCreate: Create dataset block

dmBlock* dmBlockCreate(dmDataset* ds,char* blockName,dmBlockType blockType)

(R1): This routine creates a new datablock in the given dataset. BlockType can be dmTABLE or dmIMAGE. The created block has no rows/columns, and no axes, and no header keys. This routine has been superseded by dmDatasetCreateTable and dmDatasetCreateImage.

4.2.4 dmBlockCreateCopy

dmBlock* dmBlockCreateCopy(dmDataset* ds, char* blockName, dmBlock* parent, dmBool copyData)

(R1.6): This routine creates a new datablock in the given dataset, copying its structure from the arbitrary parent DM virtual block. The new block will be created as follows:

- The block type (table or image) is inherited from the parent.
- All header keys are inherited.

- All data subspace filters are inherited, with any run time filters on the parent block becoming a permanent part of the returned block.
- If the block is a table, all table columns are inherited, but the number of rows is set to zero (unless copyData=dmTRUE)
- If the block is an image, the image dimensions are inherited, but no image data is written.

If copyData=dmTRUE, the block data will be copied at creation time. This provides a simple mechanism for en-masse block duplication.

If the caller has specified copyData=dmFALSE, dmTableCopyRow will be appropriately initialized, and the user would then be free to add or delete columns and keys from the block to complete its modified structure. The user may then start writing data to the block, the rationale being that often we want a very similar file but with somewhat different actual data, and maybe with an additional column. Using row-based I/O, one may even copy the data from a source table row by row, interspersed with new column data.

4.2.5 dmBlockCreateSubspaceCpt

int dmBlockCreateSubspaceCpt(dmBlock* block)

(R1.6): Create a new component of the data subspace of the block (ie. make a new row in the virtual data subspace table).

4.2.6 dmBlockDelete

int dmBlockDelete(dmBlock* block)

(R1.7): Deletes the specified block from the physical dataset. In the FITS kernel, if the block is a primary image array and is the only HDU in the file, you have a problem: the file is empty. If you close the dataset without creating another block or deleting the dataset, the kernel will write a null primary image (NAXIS=0) to give you a minimal valid dataset.

4.2.7 dmBlockGetCurrSubspaceCpt

long dmBlockGetCurrSubspaceCpt(dmBlock* block)

(R1.6): Get the component number (ie. the row number in the virtual data subspace table) to which the next subspace write will take place when doing dmSubspaceColCreate, dmSubspaceColSet, ect. This value is changed using dmBlockSetSubspaceCpt. If the data subspace is uninitialized, the value of this function is 1 and not 0, since any subspace create routine will create and write to the first component of the subspace if no components have been created.

4.2.8 dmBlockGetDataset

dmDataset* dmBlockGetDataset(dmBlock* block)

Given a pointer to a block, return a pointer to its parent dataset.

Context 1 (R1): The block (table or image) was opened starting from a currently open dataset. In this case you have the dataset pointer around already, but it's convenient to be able to get it just from the block, say if you passed the block pointer to a subroutine.

Context 2 (R1): The block and dataset were opened using dmDatasetTableOpen or dmImageOpen which return a block pointer. In this case, this routine is the only way the user can get the dataset pointer. Once they have it, they can then proceed to open or create other blocks using the various dmDataset routines; this is necessary in some cases, but is discouraged. If you want to access multiple blocks in the dataset, it's better to do a standard dmDatasetOpen.

4.2.9 dmBlockGetSubspaceColNo

dmDescriptor* dmBlockGetSubspaceColNo(dmBlock* block, long colNo)

(R1.6): dmBlockGetSubspaceColNo(block, n) gets the nth open subspace column for the block. This routine may be used to traverse all the subspace columns and so determine the full data subspace.

4.2.10 dmBlockGetKernelKey

char* dmBlockGetKernelKey(dmBlock* table, long n)

(R3): Return a string containing header info for the nth kernel key (for FITS, the actual header order; for other kernels, the requirement is just that calling this from 0 to dmBlockGetNoKernelKeys-1 will get you all the headers.) Diagnostic routine to bypass data model interpretation.

4.2.11 dmBlockGetKey

dmDescriptor* dmBlockGetKey(dmBlock* table, long keyNo)

(R1.5): Return the descriptor for the Nth keyword. Note that in the general case there will NOT be a one-to-one mapping between the list of header keywords as seen through the DataModel and those keywords that actually appear in the associated physical file. For example, FITS required and reserved keywords are filtered, and so are not visible within the DataModel. The keyNo in this call is the DataModel key number, not the FITS header card number.

4.2.12 dmBlockGetKeyList: Get keyword list

dmDescriptor** dmBlockGetKeyList(dmBlock* table, long* numKeys)

(R1.6): Return descriptors for all the keys in the header, and the number of keys. User must free the memory allocated for the array, but not for each descriptor.

4.2.13 dmBlockGetName

int dmBlockGetName(dmBlock* block, char *blockName, int maxlen)

(R1) Given a block pointer, copy at most maxlen characters of the block name into the pre-allocated char* blockName array.

4.2.14 dmBlockGetNo: Get block no

int dmBlockGetNo(dmBlock* block)

(R1.5): Return the number of the block in the dataset. The number is between 1 and the total number of blocks in the dataset. The numbering makes no distinction between tables and images.

4.2.15 dmBlockGetNoSubspaceCols

int dmBlockGetNoSubspaceCols(dmBlock* block)

(R1.6) Get the number of subspace columns in the data subspace.

4.2.16 dmBlockGetNoKernelKeys

long dmBlockGetNoKernelKeys(dmBlock* block)

(R3): Return the number of kernel level keys (e.g. FITS header cards, QPOE headers). Diagnostic routine to bypass data model interpretation.

4.2.17 dmBlockGetNoKeys

long dmBlockGetNoKeys(dmBlock* block)

(R1): Gets the number of header keys in the block.

4.2.18 dmBlockGetNoSubspaceCpts

int dmBlockGetNoSubspaceCpts(dmBlock* block)

Context 1: (R1.6) Get the number of components in the data subspace (always 1).
Context 2: (R1.6) Get the number of components (support more than one component)

4.2.19 dmBlockGetPrefAxes: Get preferred column or axis

dmDescriptor** dmBlockGetPrefAxes(dmBlock* block, int* ncols)

(R1.8): The routine returns the number of preferred axes (in ncols) and a list of their descriptors as the return value.

The 'preferred axes' mechanism stores the names of some subset of the table columns or image axes with the data block. These 'preferred' axes or columns are hints to generic software about the independent variables recorded in the table: they are the 'most interesting' axes and given in a specific order. For instance, a table plotting program which plots one column against a second might use the first two preferred quantities as the default choice of columns to plot; a binning program might take a spectrum table and make a 1-D image out of the first preferred axis and the preferred weight. An image display program might use the first two preferred axes of an n-dimensional image as its default choice for image display. Note that for file format compatibility reasons the 'interesting' axes are not always the first ones. Preferred axes are optional; in principle we could support any number N of preferred axes, but it is hard to see applications for the case of N more than 3, so we may introduce a 'maximum number of preferred axes supported' which should be at least 3.

The preferred axes may be implemented as header keys for the block, with a specific convention for the key names as suggested in the abstract design document. This means that the preferred axes may also be directly manipulated by the key read/write routines, and the specific routines to support them are just convenience routines; this is OK as the preferred axes are considered to be a higher layer convention and not part of the fundamental structure of the file.

4.2.20 dmBlockGetPrefWeight: Get preferred weighting column or axis

dmDescriptor* dmBlockGetPrefWeight(dmBlock* block)

(R1.8): Returns the descriptor of the 'preferred weight' axis, or null if no such axis is defined. The intent of the 'preferred weight' is a hint to generic software about the dependent variable in the table. A table with a preferred weight column is taken to represent a histogram whose values are given in the preferred weight column. For example, a PHA table might have either COUNTS or RATE as its preferred weight. A line graphics program would then know to use that as its default Y axis. At the moment there is no interpretation of a preferred weight for an image block, but calling the routine for an image block does not cause an error.

4.2.21 dmBlockGetSubspace

dmBlockGetSubspace(dmBlock* block,dmDescriptor** ddList,int* ncolss,int* ncpts)

(R1.9): Return array of subspace column descriptors for the data subspace. These routines return elements of the data subspace. The various properties of each DSS column may be accessed using the usual descriptor routines (as for table columns).

4.2.22 dmBlockGetType

dmBlockType dmBlockGetType(dmBlock* block)

(R1): Return the dmBlockType of the given block (table or image).

4.2.23 dmBlockGetTypeStr

char* dmBlockGetTypeStr(dmBlock* block)

(R2): Return a string giving the block type of the given block. Used instead of dmBlockGetType when you want to print a string with the information.

4.2.24 dmBlockInSubspace: Test whether in data subspace

dmBool dmBlockInSubspace(dmBlock* block, char** names, dmDataType* types, void* values, int n)

(R1.9): Returns true if the given 'row' of n items has values which lie in the data subspace of block.

4.2.25 dmBlockIntersectSubspace

int dmBlockIntersectSubspace(dmBlock* block1,dmBlock* block2,dmBlock* block0)

(R1.6): Set the data subspace of block0 to be the intersection of block1 and block2 (block0, block1 and block2 need not all be distinct). Run time filters on blocks 1 & 2 become a permanent part of the data subspace of block0.

4.2.26 dmBlockMoveToKey

dmDescriptor *dmBlockMoveToKey(dmBlock* block, char* name)

(R1.5): Move in header to position immediately following the key with the given name. Used to allow you to insert header entry at a specified position. The purpose of this and related routines is for the case when you copy a header from another file, or edit a file in place, and then want to insert a new key at a specific position in the header. The next dmKeyWrite will put a key immediately following the key named in this routine. Returns a pointer to the resulting descriptor, or NULL if no match.

4.2.27 dmBlockMoveToKeyNo

dmDescriptor *dmBlockMoveToKeyNo(dmBlock* block,long no)

(R1.5): Move in header to position immediately following the given numbered key. If this call is followed by dmKeyWrite, key number no+1 will be written, and following keys will be shoved down by one. Returns a pointer to the resulting descriptor, or NULL on error. Also returns NULL if `no` is larger than the largest key number, but also moves the header position to the end of the header block.

4.2.28 dmBlockOpen: Open dataset block

dmBlock* dmBlockOpen(dmDataset* ds, char* blockName)

Context 1 (R1): Raw block open. This routine opens a datablock in the given dataset. It searches for any block of the given name. If the name given is null, opens the next datablock in the dataset. If a name is given but no such datablock is found, or if there are no datablocks in the dataset, returns null. Works for either table or image blocks; use dmBlockGetType to find out which you have opened.

Context 2 (R1.7): Filtered block open. As above, but support a virtual block specification (filtered block using ASC filter syntax). The resulting virtual block has the following properties compared to the raw 'parent' block that is being filtered:

- The header keys are the same as for the parent block.
- If an image subsection, the axes and pixel range are a subset of that in the parent block.
- If a table filter, the columns and rows are a subset of that in the parent block, as specified by the filter.
- If an image created by binning the table, the axes and pixel range are as specified by the binning command.

- In all the above cases the data subspace of the virtual block is the intersection of the data subspace of the parent block and the restrictions implied by the filter.

Context 3 (R1.7): Vector column and coordinate support. Read the MTYPE and MFORM keywords to spot vectored columns, and coordinate info to spot coordinate descriptors. Create vector columns automatically when RA—TAN/DEC–TAN pairs are found.

4.2.29 dmBlockPrintKernel: List kernel block

int dmBlockPrintKernel(dmBlock* block)

(R3): Diagnostic routine to inspect the block contents at the kernel level, bypassing the layer of interpretation added by the data model.

4.2.30 dmBlockPrintSubspace: List data subspace

int dmBlockPrintSubspace(dmBlock* block)

(R2): Diagnostic routine. List the data subspace associated with the given block, printing a description of it to standard output.

4.2.31 dmBlockReadComment

int dmBlockReadComment(dmBlock* block, char** tag, char** comment)

(R1.5) Read comment from the header at the current position. There may be more than one comment (eg, both COMMENT and HISTORY) at a given position (acceptable tag values are dmHISTORY and dmCOMMENT). If there is no comment at the current position, return dmFALSE.

Example program to read all header keys and comments in a block:

```
char* tag;
char* comment;
dmDescriptor* key;
char name[MAXLEN];

int n = dmBlockGetNoKeys( block );

dmBlockMoveToKeyNo( block, 0 );
/* Get initial block comments (if any) */
while (dmBlockReadComment( block, &tag, &comment )) {
  printf( "%s %s\n", tag, comment );
```

```
  free( tag );
  free( comment );
}
for ( i=1; i<=n; i++ ) {
  key = dmBlockGetKey(block, i);
  dmGetName( key, name, MAXLEN );
  printf("KEY %d: %s\n", i, name);
  /* Get block comments for key */
  while (dmBlockReadComment( block, &tag, &comment )) {
    printf( "%s %s\n", tag, comment );
    free( tag );
    free( comment );
  }
}
```

4.2.32 dmBlockSetPrefAxes

int dmBlockSetPrefAxes(dmBlock* block,dmDescriptor** axes,int naxes)

(R1.8): Set the preferred axes for an image or a table. The input arguments include an array of dmDescriptor pointers and the size of that array. See the description of dmBlockGetPrefAxes. See also dmBlockSetPreferred. In the QPOE kernel, sets the QPOE keys if naxes is 2.

4.2.33 dmBlockSetPrefWeight

int dmBlockSetPrefWeight(dmBlock* block, dmDescriptor* weight)

(R1.8): Set the preferred weight column for a table. See the description of dmBlockGetPrefWeight. See also dmBlockSetPreferred.

4.2.34 dmBlockSetPreferred

int dmBlockSetPreferred(dmBlock* block,char* spec)

(R1.8): Set the preferred axes and weight for an image or a table, using a string which will be parsed by the library. This routine parses the string and calls dmBlockSetPrefAxes and dmBlockSetPrefWeight; it is provided as a convenience to make code more readable.

The syntax of the SPEC argument is: "name1,name2,...nameN" or "weight(name1,name2,...nameN)"; for example "DETPOS,TIME" or "PHA(DETPOS,TIME)". The comma-separated list of names are the names of the preferred columns or axes to be passed to SetPrefAxes. If this list is enclosed in parentheses and preceded by another name, that name is the weight to be passed to SetPrefWeight.

The list may be null, e.g. "PHA()". This syntax is meant to encapsulate the idea of the preferred axes as a default interpretation of the table (or image) as a function of several variables whose value may be the image data values, the histogram of table values (event list interpretation) or the specified weight column (histogram table interpretation).

4.2.35 dmBlockSetSubspaceCpt

int dmBlockSetSubspaceCpt(dmBlock* block, int cptNo)

(R1.6): Change the number of the current subspace component (component must already exist). Return an error if the component number is less than one or greater than the maximum component for the block.

4.2.36 dmBlockWriteComment

int dmBlockWriteComment(dmBlock* block, char* tag, char* comment)

(R1): Write comment to the header at the current header position. A comment is not a header key, but rather text which has a defined position relative to the header keys. Acceptable tag values are dmHISTORY and dmCOMMENT.

4.3 dmColumn Routines

4.3.1 dmColumnCreate

dmDescriptor* dmColumnCreate(dmBlock* table, char* name, dmDataType type, long slen, char* unit, char* desc)

(R1): Create a scalar column, specifying its name, data type, unit and element dimension. The element type, element dimensionality and array dimensions are given their default values. The `slen` argument gives the length of the string for a string column; it is ignored (but still required) for other data types.

4.3.2 dmColumnCreateArray

dmDescriptor* dmColumnCreateArray(dmBlock* table, char* name, dmDataType type, long slen, char* unit, char* desc, long size)

(R1): Create a 1D array column, specifying its axis dimension. The descriptor has element dimension 1, array dimension 1, and element type Value. It differs from a scalar column in that there is more than one element in each cell; the number of elements is given by the `size` argument. It is a special case of an ND array.

4.3.3 dmColumnCreateInterval

> dmDescriptor* dmColumnCreateElement(dmBlock* table, char* name, dmDataType type, char* unit, char* desc, char** cptNames, dmElementType elementType)

R1.7 Note change to calling sequence since last rev.

Context 1 (R2): Create a scalar interval column, specifying its element type. The different element types are described in the abstract design document. The default element type is dmVALUE, which has one value per scalar element. Calling this routine with element type dmVALUE is equivalent to calling dmColumnCreate. dmVALUE is a ‘simple element type’ with only one component. All other element types are ‘compound element types’ with multiple components.

Context 2 (R2.1): Support element types dmINTERVAL and dmRANGE, which have a max and min value.

Context 3 (R2.1): Support further element types. (Add text here to define compound element types and the order of their components).

4.3.4 dmColumnCreateNDArray

> dmDescriptor* dmColumnCreateNDArray(dmBlock* table, char* name, dmDataType type, char* unit, char* desc, long* axes, long naxes)

(R2.0): Create an n-dimensional array column, including its array specification. Each cell of the table will contain an n-dimensional array of scalar elements. The dimensionality is given by naxes, and the length of each axis is given in the axes array.

4.3.5 dmColumnCreateVarArray

> dmDescriptor* dmColumnCreateVarArray(dmBlock* table, char* name, dmDataType type, char* unit, char* desc, long size)

(R1.8): Special routine to create a 1D array column, which will be stored as a variable-length array if the underyling kernel provides such support. The value of size is either zero or the maximum allowed length of any row of the data. This routine is provided to support the FITS variable length array facility. Functionally the column is the same as that created by dmColumnCreateArray, but the kernel is encouraged to store it in a disk-space efficient manner. String columns are not supported in this version.

4.3.6 dmColumnCreateVector

> dmDescriptor* dmColumnCreateVector(dmBlock* table, char* name, dmDataType type, char* unit, char* desc, char** cptNames, long dim)

(R1.7): Create a descriptor for a vector column, specifying its name, data type, unit, component names and element dimensionality. The element type is Value and the array dimensionality is 0. The difference from a scalar column is that a vector column may have element dimensionality greater than 1, with associated component names for each component of the vector. In FITS, it is implemented using separate columns for each component.

4.3.7 dmColumnCreateGeneric

dmDescriptor* dmColumnCreateGeneric(dmBlock* table, char* name, dmDataType type, long slen, char* unit, char* desc, dmElementType elementType, char** cptNames, long dim, long* axes, long naxes)

(R2.1): Create a generic column, allowing for vector and array specifications and interval type specification. This routine may be used to create more complicated structures such as vectored arrays of dmINTERVAL elements. The component names for a vectored compound element type are given in an order such that all the compound element components for each vector component are given together, in the order specified under dmColumnCreateElement.

4.3.8 dmColumnGetNo

long dmColumnGetNo(dmDescriptor* dd)

Context 1 (R1): Return the number of the column, given its descriptor.
Context 2 (R1): Return zero if descriptor is not a column.

4.3.9 dmColumnInsertAfter

int dmColumnInsertAfter(dmBlock* table, char* name)

(R3): The next call to a createColumn routine will create a column with column number one greater than the column with the given name. All columns with equal or greater column number will have their column numbers incremented (i.e. moved to the right). This routine must be called prior to writing the first data row or an error will be generated.

4.3.10 dmColumnInsertNo: Insert column by number

int dmColumnInsertNo(dmBlock* table, long colNo)

(R3): The next call to a createColumn routine will create a column with the given column number. Other columns will be moved to the right to make room. This routine must be called prior to writing the first data row or an error will be generated.

4.4 dmCoord Routines

4.4.1 dmCoordCalc

int dmCoordCalc_s(dmDescriptor* dd, short* value, double* result)
int dmCoordCalc_l(dmDescriptor* dd, long* value, double* result)
int dmCoordCalc_f(dmDescriptor* dd, float* value, double* result)
int dmCoordCalc_d(dmDescriptor* dd, double* value, double* result)
int dmCoordCalc_ub(dmDescriptor* dd, char* value, double* result)
int dmCoordCalc_us(dmDescriptor* dd, unsigned short* value, double* result)
int dmCoordCalc_ul(dmDescriptor* dd, unsigned long* value, double* result)

Context 1 (R1.8): Return the coordinate value corresponding to an input value. The type of the input value is up to the user; the type of the coordinate value is always double. One set of routines is provided for both scalar and vector descriptors, so the input and return arguments are pointers. The input descriptor's parent is a table column.

Context 2 (R1.8): The input descriptor's parent is an image axis group.

Context 3 (R2.0): The input descriptor's parent is an image data descriptor (the coordinate here rescales the image values, rather than laying a coord grid on the image).

These routines are provided to get coordinate values for input values not in the table (image). To read the columns of a table returning the coordinate values, just use the normal column read routines using the coordinate quantity's data descriptor instead of the raw column's data descriptor.

Example:

```
dmDescriptor* pos = dmTableOpenColumn( table, "POS" );  /* Vector column, long datatype */
dmDescriptor* world = dmGetCoord( pos );
long value[2] = { 14, 38 };
double result[2];
dmCoordCalc_l( world, value, result );
dmDescriptor* pha = dmTableOpenColumn( table, "PHA" );  /* Array scalar column, short datatype */
dmDescriptor* energy = dmGetCoord( pha );
double value2 = 4.8;  /* Even though the column is integral */
double result2;
dmCoordCalc_d( energy, &value2, &result2 );
```

4.4.2 dmCoordCalcInteger

int dmCoordCalcInteger_s(dmDescriptor* dd, short value, long* result)
int dmCoordCalcInteger_l(dmDescriptor* dd, long value, long* result)
int dmCoordCalcInteger_ub(dmDescriptor* dd, char value, long* result)
int dmCoordCalcInteger_us(dmDescriptor* dd, unsigned short value, long* result)
int dmCoordCalcInteger_ul(dmDescriptor* dd, unsigned long value, long* result)

(R2.0) Calculate for integer coordinate transforms. We will eventually add routines to support lookup calculations; for now, the lookup table must be opened directly. Given the pixel value, calculate the transformed value and return it as a long.

4.4.3 dmCoordInvert

int dmCoordInvert_s(dmDescriptor* dd, double* value, short* result)
int dmCoordInvert_l(dmDescriptor* dd, double* value, long* result)
int dmCoordInvert_f(dmDescriptor* dd, double* value, float* result)
int dmCoordInvert_d(dmDescriptor* dd, double* value, double* result)
int dmCoordInvert_ub(dmDescriptor* dd, double* value, char* result)
int dmCoordInvert_us(dmDescriptor* dd, double* value, unsigned short* result)
int dmCoordInvert_ul(dmDescriptor* dd, double* value, unsigned long* result)

(R1.8) Apply the inverse coordinate transform, mapping from world coordinates to pixel coordinates. If the inverse mapping is not meaningful or single-valued, or if the value is outside the range of the mapping, an error code is returned.

4.4.4 dmCoordCreate

dmDescriptor* dmCoordCreate_s(dmDescriptor* dd, char* name, char* unit, char** cptnames, long dim, char* transform, short* crpix, double* crval, double* cdelt, double* parameters)
dmDescriptor* dmCoordCreate_l(dmDescriptor* dd, char* name, char* unit, char** cptnames, long dim, char* transform, int* crpix, double* crval, double* cdelt, double* parameters)
dmDescriptor* dmCoordCreate_f(dmDescriptor* dd, char* name, char* unit, char** cptnames, long dim, char* transform, float* crpix, double* crval, double* cdelt, double* parameters)
dmDescriptor* dmCoordCreate_d(dmDescriptor* dd, char* name, char* unit, char** cptnames, long dim, char* transform, double* crpix, double* crval, double* cdelt, double* parameters)
dmDescriptor* dmCoordCreate_ub(dmDescriptor* dd, char* name, char* unit, char** cptnames, long dim, char* transform, char* crpix, double* crval, double* cdelt, double* parameters)
dmDescriptor* dmCoordCreate_us(dmDescriptor* dd, char* name, char* unit, char** cptnames, long dim, char* transform, unsigned short* crpix, double* crval, double* cdelt, double* parameters)
dmDescriptor* dmCoordCreate_ul(dmDescriptor* dd, char* name, char* unit, char** cptnames, long dim, char* transform, unsigned long* crpix, double* crval, double* cdelt, double* parameters)

Context 1 (R1.7): Create a coordinate transform object and associate it with an existing column data descriptor. Different routines are provided for different data descriptor types; the coordinate quantity must always be of type double. Using the dmGetScalar routines but on the coordinate data descriptor will return the coordinate data values.

The arguments are:

- dd: the parent descriptor.
- name: name of the coordinate descriptor
- unit: unit of the coordinate descriptor.
- cptnames: names of coordinate components for 2-dim coordinates (e.g. 'RA', 'DEC'). Null for 1-D case.
- dim: the element dimension of the coordinate descriptor which must be the same as the parent descriptor, or an error is set.
- transform: the type of the transform. If null or blank, the value "LINEAR" is assumed. Another supported value is "TAN". Other WCS transforms will be supported later. However, no check is made by this routine to see if the transform is known.
- crpix: the reference value x0 of the parent descriptor. This is an array with dim elements.
- crval: the corresponding value y0 of the coordinate descriptor. This is also an array.
- cdelt: the transform scale: dy/dx at x=x0, array with dim elements.
- parameters: Additional transform parameters, usually null. The number of parameters depends on the transform.

Context 2: (R1.7) As above, but parent descriptor is an image axis (created by dmArrayCreateAxisGroup) for an image block.

Context 3: (R2.0) As above, but parent descriptor is an image data descriptor.

Context 4: (R2.0) As above, but parent descriptor is a key.

Context 5: (R3.0) As above, but parent descriptor is an image axis for an array column.

4.4.5 dmCoordCreateInteger

dmDescriptor* dmCoordCreateInteger_s(dmDescriptor* dd, char* name, char* unit, long dim, short crpix, long crval, long cdelt)
dmDescriptor* dmCoordCreateInteger_l(dmDescriptor* dd, char* name, char* unit, long dim, long crpix, long crval, long cdelt)
dmDescriptor* dmCoordCreateInteger_ub(dmDescriptor* dd, char* name, char* unit,

long dim, char crpix, long crval, long cdelt)
dmDescriptor* dmCoordCreateInteger_us(dmDescriptor* dd, char* name, char* unit, long dim, unsigned short crpix, long crval, long cdelt)
dmDescriptor* dmCoordCreateInteger_ul(dmDescriptor* dd, char* name, char* unit, long dim, unsigned long crpix, long crval, long cdelt)

(R2.0) Create an integer-valued 1D coord transform object (only available for integer- valued data descriptors). The only transform allowed is the linear transform. This is useful for retaining the original coordinate system for blocked images and rebinned PHA spectra.

4.4.6 dmCoordCreateLinear

dmCoordCreateLinear_s(dmDescriptor* dd, char* name, char* unit, short crpix, double crval, double cdelt)
dmCoordCreateLinear_l(dmDescriptor* dd, char* name, char* unit, long crpix, double crval, double cdelt)
dmCoordCreateLinear_f(dmDescriptor* dd, char* name, char* unit, float crpix, double crval, double cdelt)
dmCoordCreateLinear_d(dmDescriptor* dd, char* name, char* unit, double crpix, double crval, double cdelt)
dmCoordCreateLinear_us(dmDescriptor* dd, char* name, char* unit, unsigned short crpix, double crval, double cdelt)
dmCoordCreateLinear_ul(dmDescriptor* dd, char* name, char* unit, unsigned long crpix, double crval, double cdelt)
dmCoordCreateLinear_ub(dmDescriptor* dd, char* name, char* unit, unsigned char crpix, double crval, double cdelt)

(R1.7) This wraps the simple, common case of a linear coord transform of dimension 1 so you can use literal constants instead of having to create a variable with pointers, e.g.

```
dmCoordCreateLinear_l( pha, "Energy", "keV", 0, 0.15, 0.01 );
```

4.4.7 dmCoordCreateLookup

dmDescriptor* dmCoordCreateLookup(dmDescriptor* dd, char* name, char* unit, char* transform, char* resultType)

(R1.7) Renamed routine.
(R2.1) Create a lookup coordinate transform. The transform argument is a virtual datablock specification of a table which contains two columns, the first of type equal to the data descriptor

type and the second of type equal to the resultType argument. The table is assumed to be sorted on the first column. The table data descriptors will determine the type of interpolation to be carried out for numeric types. This type of transform also supports string types (e.g. integer to string lookup, or vice versa).

4.4.8 dmCoordGetParams

double* dmCoordGetParams(dmDescriptor* dd, long* nparams)

(R1.7) Return the coordinate transform parameters, and set nparams to the number of them. These are special parameters needed by specific function transforms, and do not include the standard CRPIX, CRVAL, CDELT values. In fact, nparams is almost always zero.

4.4.9 dmCoordGetParent

dmDescriptor* dmCoordGetParent(dmDescriptor* dd)

(R1.7) Get the descriptor to which the coordinate is attached. Inverse of dmDescriptorGetCoord. Returns null if the descriptor is not a coordinate descriptor.

4.4.10 dmCoordGetTransform

dmCoordGetTransform_s(dmDescriptor* dd, short* crpix, double* crval, double* cdelt, long dim)
dmCoordGetTransform_l(dmDescriptor* dd, long* crpix, double* crval, double* cdelt, long dim)
dmCoordGetTransform_f(dmDescriptor* dd, float* crpix, double* crval, double* cdelt, long dim)
dmCoordGetTransform_d(dmDescriptor* dd, double* crpix, double* crval, double* cdelt, long dim)
dmCoordGetTransform_ub(dmDescriptor* dd, un. char* crpix, double* crval, double* cdelt, long dim)
dmCoordGetTransform_us(dmDescriptor* dd, un. short* crpix, double* crval, double* cdelt, long dim)
dmCoordGetTransform_ul(dmDescriptor* dd, un. long* crpix, double* crval, double* cdelt, long dim)

(R1.7): Return the coordinate transform values CRPIX, CRVAL, CDELT. The array of size dim is allocated by the user (since it's usually only of size 1 or 2 and its size is determined by the dimension of the parent descriptor). The value dim is passed by the user to the routine for safety: at most dim values will be returned.

4.4.11 dmCoordGetTransformType

int dmCoordGetTransformType(dmDescriptor* dd, char* transform, long* nparams)

(R1.7): Return the transform type for a coordinate transform. The input argument may be either the original DD or the coordinate quantity DD. The number of transform parameters is returned; this is usually zero.

4.4.12 dmCoordSetTransform

dmCoordSetTransform_s(dmDescriptor* dd, short* crpix, double* crval, double* cdelt, long dim)
dmCoordSetTransform_l(dmDescriptor* dd, long* crpix, double* crval, double* cdelt, long dim)
dmCoordSetTransform_f(dmDescriptor* dd, float* crpix, double* crval, double* cdelt, long dim)
dmCoordSetTransform_d(dmDescriptor* dd, double* crpix, double* crval, double* cdelt, long dim)
dmCoordSetTransform_ub(dmDescriptor* dd, un. char* crpix, double* crval, double* cdelt, long dim)
dmCoordSetTransform_us(dmDescriptor* dd, un. short* crpix, double* crval, double* cdelt, long dim)
dmCoordSetTransform_ul(dmDescriptor* dd, un. long* crpix, double* crval, double* cdelt, long dim)

(R1.7): Change the transform values for a coord transform.

4.5 dmDataset Routines

4.5.1 dmDatasetAccess (Access dataset)

dmBool dmDatasetAccess(char* dsname, char* mode)

(R1) Check a dataset is accessible, returning either dmTRUE or dmFALSE. Supported access modes are 'R' for read only permission and 'RW' for read/write. This routine is used prior to opening to test whether a dataset already exists. It does not support filtering.

4.5.2 dmDatasetAdvanceBlocks (Move within a dataset)

dmBlock* dmDatasetAdvanceBlocks(dmDataset* ds, int RelNo)

(R1): Move a given number of datablocks relative to the current datablock, and open the corresponding datablock, returning a pointer to it. Analogous to the movrelHdu call in CFITSIO, but remember that (R3) one ASC table may correspond to more than one kernel table. RelNo may be positive or negative. If the block number obtained by adding RelNo to the current block number is outside the range of valid block numbers, the routine returns null and sets an error status. If RelNo = +1, the behaviour of the routine is the same as dmDatasetNextBlock.

4.5.3 dmDatasetClose (Close dataset)

int dmDatasetClose(dmDataset* ds)

(R1) Close an ASC dataset, flushing all buffers, closing the file(s) on disk, and freeing associated memory. Returns zero on success. Note that in the general case open blocks should be closed first.

4.5.4 dmDatasetCreate (Create dataset)

dmDataset* dmDatasetCreate(char* datasetName)

Context 1: (R1) This routine creates a new, empty dataset using the current creation kernel, and returns a pointer to the dataset (null on failure). The datasetName must be a simple dataset name without filtering.

Context 2: (R3?): See Future Requirements - Modified IRAF Kernel.

4.5.5 dmDatasetCopy (Copy dataset)

int dmDatasetCopy(char* datasetName, char* outputName)

Context 1: (R2) Makes a straight copy of an entire dataset.

Context 2: (R2) Support filtering; this routine would then reduce the COPY tool to one line.

This routine should be reviewed to see if it is needed.

Example:

```
dmDatasetCopy( "foo.fits[events][pha=4:5]", "filtered_events.fits" );
```

4.5.6 dmDatasetDelete (Delete dataset)

int dmDatasetDelete(dmDataset* ds)

Context 1 (R1): Deletes the specified physical dataset. This routine will close the dataset, release the associated memory, and delete all associated files from the disk. If the dataset is a virtual (filtered) dataset, there's nothing on disk to delete (i.e. it doesn't go and delete the underlying unfiltered dataset). This routine would normally be used when you've created a temporary dataset during the program.

4.5.7 dmDatasetGetBlockName

int dmDatasetGetBlockName(dmDataset* ds,long blockNo,char *blName,int maxlen)

(R1): Return the block name for the numbered block in the dataset, effectively letting us determine the block name before we formally open it.

4.5.8 dmDatasetGetBlockType

dmBlockType dmDatasetGetBlockType(dmDataset* ds, int block_no)

(R1): Return the block type for the numbered block in the dataset, effectively letting us determine the block type before we formally open it.

4.5.9 dmDatasetGetCurrentBlockNo (Get current block number)

int dmDatasetGetCurrentBlockNo(dmDataset* ds)

(R1) We introduce the idea of a 'current block number'. This number is that of the most recently opened block. It is changed by the OpenBlock, MoveToBlock, NextBlock or AdvanceBlock routines. The GetCurrentBlockNo routine reports this current block number. This routine and its relatives are useful when cycling through all the blocks in a dataset. Returns zero if called before any blocks have been read.

4.5.10 dmDatasetGetKernel

int dmDatasetGetKernel(dmDataset* ds, char *kerName, int maxlen)

(R1.6) Return the kernel (ie, file format type) used by the dataset.

4.5.11 dmDatasetGetName

int dmDatasetGetName(dmDataset* ds, char* name, int maxlen)

(R1.6) Return the on-disk name of the given dataset.

4.5.12 dmDatasetNextBlock: Advancing the dataset block pointer

dmBlock* dmDatasetNextBlock(dmDataset* ds)

(R1): Open the next block in the dataset. Uses the 'current block pointer'; increments the block pointer and opens the next block. Returns null if there are no more blocks in the dataset. This routine is the same as dmDatasetAdvanceBlocks(ds,+1) and is provided for convenience.

4.5.13 dmDatasetGetNoBlocks (Get number of blocks)

int dmDatasetGetNoBlocks(dmDataset* ds)

Context 1 (R1): Return number of datablocks in the dataset. This requires the library to fully scan the dataset, parsing each block header to determine the structure of the file; the number of blocks cannot in general be found without reading all the headers.

Context 2 (R3): As above, but only include true datablocks; for example, omit GTI or other kernel-level blocks recognized as part of another DM-level datablock.

4.5.14 dmDatasetMoveToBlock (Direct access to block)

dmBlock* dmDatasetMoveToBlock(dmDataset* ds, int BlockNo)

Open a block within the dataset, specifying the block by its number. The blocks are ordered in the dataset starting at 1 and going to N = dmDatasetGetNoBlocks(ds). Calling this routine with a block number outside this range will return a null value, and set an error status.

4.5.15 dmDatasetOpen (Open dataset)

dmDataset* dmDatasetOpen(char* datasetName)

Context 1 (R1): Open an existing dataset and return a pointer to a dmDataset object. The argument is the name of a dataset. If this dataset does not exist or the attempt to open fails, a null pointer will be returned. The default access mode is 'RW' (read/write), but no change will be made to the dataset if no explicit write calls are made. The dataset may be either: - a FITS file, OR - a directory containing IRAF QPOE/IMH files, OR - a single IRAF QPOE/IMH file

The routine checks each supported kernel in turn, you don't have to know in advance which kind of file it is.

Context 2 (R3?): See Future Requirements - Modified IRAF Kernel.

4.5.16 dmDatasetPrint (List dataset)

int dmDatasetPrint(dmDataset* ds, char* options)

List the blocks in a dataset to standard output. Depending on options, list details of the structure of each table. This routine is provided to allow easy diagnostics, and will normally be used as part of the dataset-dump tool. It is different from the Observation Browser since its normal use will be to dump to an ASCII file which the user will then use an editor or Unix tools to peruse. Valid options are:

- "BLOCKS": (Context 1, R2) One line per block, giving block number, block name, block type, and block shape. For tables, block shape is no of columns x no of rows (eg "Table 14 x 104122"). For images, block shape is image dimensions and data type (e.g. "Real 3D Image 512 x 512 x 3")
- "HEADER": (Context 2, R2) For each block, list all keys with one line per key giving name, value and description.
- "DESCRIPTORS": (Context 3, R3) For each block, list all descriptors: first keys, then filters, then columns, with image axes and coordinates listed together with their parent descriptors.
- "FULL": (Context 4, R3) For each block, list all keys as above, then dump column or image values in a format similar to FDUMP.

4.5.17 dmDatasetPrintKernel: List kernel dataset

int dmDatasetPrintKernel(dmDataset* ds)

(R2): Summarize the structure of the underlying dataset. If the underlying file is a FITS file, write a summary of the HDUs in the file, including their types, names, and record offsets. Do a similar thing for other kernels. Used as a diagnostic tool; if the data model is being too clever and getting confused, you can inspect the underlying file structure more directly with this routine.

4.5.18 dmDatasetRename (Rename dataset)

int dmDatasetRename(char* datasetName, char* newName)

(R2): Rename a dataset (no virtual spec allowed; no kernel changing). This version does not necessarily open the dataset. It just changes the name on disk.

4.5.19 dmDatasetCreateImage

dmBlock* dmDatasetCreateImage(dmDataset* ds, char* name, long* axes, long naxes)

(R1.7) Create new empty image in a dataset with the specified dimensions.

4.5.20 dmDatasetCreateTable

dmBlock* dmDatasetCreateTable(dmDataset* ds, char* name)

(R1.7) Create new empty table in a dataset.

4.6 dmDescriptor Routines

4.6.1 dmDescriptorCompare

dmBool dmDescriptorCompare(dmDescriptor* dd1, dmDescriptor* dd2, long mask)

(R1.6) Compare the two input descriptors, returning dmTRUE if they are the same and dmFALSE otherwise. The user specifies which descriptor fields to consider in the comparison using the mask parameter. To do this, the following pre-defined values are available to the user:

Macro	Meaning
dmDESCNAME	The name of the descriptor.
dmDESCVALUE	The value of the descriptor (descriptors must also have same dimensions and axis lengths to return dmTRUE).
dmDESCUNIT	The unit of the descriptor.
dmDESCKIND	The type of descriptor (dmKEY, dmSUBSPACE, etc.).
dmDESCTYPE	The datatype of the descriptor.
dmDESCELMTYPE	The element type of the descriptor (dmVALUE, dmRANGE, etc.)
dmDESCDIM	The dimension of the descriptor.
dmDESCLENS	The axis lengths of the descriptor.
dmDESCELMDIM	The element dimension of the descriptor.
dmDESCCPTNAMES	The component names of the descriptor.
dmDESCCMNT	The comment of the descriptor.
dmDESCALL	All of the above fields.

These fields can be combined using bit operators to achieve any desired combiniation. For example, to compare the values and comments of two descriptors, set mask to dmDESCVALUE & dmDESCCMNT. To compare all fields save the units, set mask to dmDESCALL & ~dmDESCUNIT. All comparisons between text values (units, comments, names, ect.) are case insensitive.

Context 1 (R1.6): Only scalar keys are fully supported. Although dmDescriptorCompare will not automatically fail for other descriptor types, comparisons may not be done intelligently.

4.6.2 dmDescriptorDelete

int dmDescriptorDelete(dmDescriptor* dd)

(R1.5) Deletes from the physical block (and dataset) the specified descriptor object (e.g., column or keyword).

4.6.3 dmDescriptorGetCoord

dmDescriptor* dmDescriptorGetCoord(dmDescriptor* dd)

Context 1 (R1.7): Return the first coordinate descriptor associated with a given descriptor. Return null if there is no such descriptor.

4.6.4 dmDescriptorGetCoordNo

dmDescriptor* dmDescriptorGetCoordNo(dmDescriptor* dd, long no)

(R1.7): Return the nth coord object associated with a descriptor.

4.6.5 dmDescriptorGetNoCoords

int dmDescriptorGetNoCoords(dmDescriptor* dd)

(R1.7): Get number of coords associated with the descriptor.

4.6.6 dmDescriptorGetRange

int dmDescriptorGetRange_s(dmDescriptor* dd, short* min, short* max)
int dmDescriptorGetRange_l(dmDescriptor* dd, long* min, long* max)
int dmDescriptorGetRange_f(dmDescriptor* dd, float* min, float* max)
int dmDescriptorGetRange_d(dmDescriptor* dd, double* min, double* max)

Get the overall range of values associated with the given descriptor.

(R1.6): Supported for columns only. These functions do no checking of the actual values stored in a column but rather look for keywords which describe the columns range (for instance, TLMIN and TLMAX keys). For double type columns called time we recognize the possibility of special syntax (i.e. TSTART and TSTOP keys).

4.6.7 dmDescriptorGetType

dmDescriptorType dmDescriptorGetType(dmDescriptor* dd)

(R1): Return the dmDescriptorType of the specified descriptor (dmKEY, dmCOLUMN, dmSUBSPACE, etc.).

4.6.8 dmDescriptorInfo: Get all descriptor info

dmDescriptorInfo(dmDescriptor* dd, char* name, char* type, char* unit, char* desc, char* elementType, char*** cptNames, long* dim, long** axes, long* naxes)

(R2) An alternate way of getting the descriptor info, retrieving all descriptor attributes at once. *Note changes to calling sequence 1997 Jul 20.*

4.6.9 dmDescriptorIs3SigLimit

int dmDescriptorIs3SigLimit(dmDescriptor* dd)

(R3.0): Tests whether the (current row) value of the given descriptor is a 3 sigma upper limit (depending on an internal uncertainty level parameter). Routines to handle uncertainty level specification are not yet defined. Until we support this, this routine is the same as dmDescriptorIsUpperLimit.

4.6.10 dmDescriptorIsUpperLimit

int dmDescriptorIsUpperLimit(dmDescriptor* dd)

(R3): Tests whether the (current row) value of the given descriptor is an upper limit: specifically, whether the MIN value of the element is zero or less. Only works for a scalar descriptor.

4.6.11 dmDescriptorPrintList: List descriptors

int dmDescriptorPrintList(dmBlock* table)

(R3): List all data descriptors and their properties (high level debug diagnostic routine).

4.6.12 dmDescriptorSetRange

int dmDescriptorSetRange_s(dmDescriptor* dd, short min, short max)
int dmDescriptorSetRange_l(dmDescriptor* dd, long min, long max)
int dmDescriptorSetRange_f(dmDescriptor* dd, float min, float max)
int dmDescriptorSetRange_d(dmDescriptor* dd, double min, double max)

Set the overall range of values associated with the given descriptor.
(R1.6): Supported for columns only. Performing this operation is the equivalent of writing TLMIN and TLMAX keys for the column.

4.7 dmGet Routines

These generic routines operate on descriptors; they should really be called dmDescriptorGetxxx for consistency, but for once we decided to let brevity win since they are the core routines of the library.

4.7.1 dmGetArray: Get array value

```
int dmGetArray_s(dmDescriptor* dd, short *values, long npixels)
int dmGetArray_l(dmDescriptor* dd, long *values, long npixels)
int dmGetArray_f(dmDescriptor* dd, float *values, long npixels)
int dmGetArray_d(dmDescriptor* dd, double *values,long npixels)
int dmGetArray_q(dmDescriptor* dd, int *values,long npixels)
int dmGetArray_c(dmDescriptor* dd, char **values,long npixels)
int dmGetArray_br(dmDescriptor* dd,char **values, long npixels)
int dmGetArray_ub(dmDescriptor* dd, unsigned char* values, long npixels)
int dmGetArray_us(dmDescriptor* dd, unsigned short* values, long npixels)
int dmGetArray_ul(dmDescriptor* dd, unsigned long* values, long npixels)
```

Reads all of the data in an array.

Context 1 (R1): Read value of an array column data descriptor, returning the values into the pre-allocated 1-dimensional values array (of size npixels), ignoring the fact that the array may actually be a higher-dimensional object.

Context 2 (R1): Results are undefined if the data type of the descriptor does not match that of the routine (it would be too inefficient to type cast an entire array).

Context 3 (R2): Read value of an image data descriptor (i.e. return all the pixel values of the data in an image).

Context 4 (R2): Read value of an array key descriptor.

4.7.2 dmGetArrayDim

```
long dmGetArrayDim(dmDescriptor* dd)
```

dmGetArrayDim returns the descriptor's array dimensionality.

Context 1 (R1): Scalar or vector descriptor (column, key, image data, filter, coord, image axis), return value of zero.

Context 2 (R1): 1-d array descriptor (column, key, image data, filter), return value of one.

Context 3 (R1.7): N-d array column or image data descriptor, return value of N. This is the general case.

4.7.3 dmGetArrayDimensions

```
long dmGetArrayDimensions(dmDescriptor* dd, long **axislengths)
```

Context 1 (R1): dmGetArrayDimensions, most useful for multi-dimensional arrays, returns the length of each axis of the array in axislengths, and also returns the value of dmGetArrayDim (which is the size of the axislengths array) as the function value. It is the caller's responsibility to free the memory allocated to *axisLengths. Works for all flavors of descriptor.

4.7.4 dmGetArraySize

long dmGetArraySize(dmDescriptor* dd)

Returns the total number of elements in the array (most useful for 1-D arrays)

Context 1 (R1): Scalar or vector descriptor (column, key, image data, filter, coord, image axis): returns 1.

Context 2 (R1): 1-d array descriptor (column, key, image data, filter): returns number of array elements.

Context 3 (R1.7): N-d array descriptor (column, image data): returns total number of elements found by multiplying all axis dimensions together. This is the general case.

4.7.5 dmGetCpt

dmDescriptor* dmGetCpt(dmDescriptor* dd, long cptNo)

Context 1 (R1.7): Return a descriptor of scalar type corresponding to one component of a vector column descriptor. Calls to dmGetScalar, etc, on this descriptor will then work to access the specified component of the vector column only. If cptNo is more than the number of components, return null.

Context 2 (R1.7): If the element dimension of the argument is 1 (dd is a scalar descriptor), return the descriptor itself.

Context 3 (R1.7): If the argument is a coordinate descriptor of dimension more than 1, return null, since getting one component of a 2D transform is a bad idea.

4.7.6 dmGetCptName

int dmGetCptName(dmDescriptor* dd, long cptNo, char *cptName, int maxlen)

Return the component name for the Nth vector component of the descriptor. The number of components may be found using dmGetElementDim.

Context 1 (R1.7): For scalar descriptors (M=1) the component name is required to be identical to the descriptor name. The routine is thus equivalent to dmGetName for cptNo=1, and sets an error for other values of cptNo.

Context 2 (R1.7): For vector descriptors (M>1) the routine returns the component name for the given component number. For a component number which is out of range, or for a null descriptor, returns a null string and sets an error condition.

4.7.7 dmGetDataType

dmDataType dmGetDataType(dmDescriptor* dd)

(R1): Get the data type of a descriptor.

4.7.8 dmGetDataTypeName

char* dmGetDataTypeName(dmDescriptor* dd)

(R2): Get a string giving the data type of a descriptor, suitable for printing.

4.7.9 dmGetDesc

int dmGetDesc(dmDescriptor* dd, char *description, int maxlen)

(R1) Get the 'description' text for a descriptor.

4.7.10 dmGetDisp: Get descriptor display format

int dmGetDisp(dmDescriptor* dd, char *display, int maxlen)

(R1.9) Get the display format hint for a descriptor. This string is a Fortran-style format compatible with the FITS TDISP keyword. It is used as a hint to generic software for a suitable display format for the data.

4.7.11 dmGetElement: Get compound element/Element value

int dmGetElement_s(dmDescriptor* dd, dmElementType type, short* result, long maxlen)
int dmGetElement_l(dmDescriptor* dd, dmElementType type, long* result, long maxlen)
int dmGetElement_f(dmDescriptor* dd, dmElementType type, float* result, long maxlen)
int dmGetElement_d(dmDescriptor* dd, dmElementType type, double* result, long maxlen)

Note (R1.7): Spec for this routine changed.

Context 1 (R2.1): Read compound value of a column data descriptor when the descriptor has the same element type as the input argument element type. Return in the result array an array of 0, 1, 2 or 3 values depending on the element type. The argument maxlen is used to pass in the declared length of the result array, which is checked agains the number of values returned for safety.

Context 2 (R2.1): Force element type of result to be that of the input argument, when descriptor has a different element type.

Context 3 (R2.1): Support key descriptors.

Context 4 (R2.1): Support filter descriptors.

4.7.12 dmGetElementDim

long dmGetElementDim(dmDescriptor* dd)

Returns the number of vector components for the descriptor. In the ASC data model, there are three levels of dimensionality: the N-dimensional array of elements in a table cell or in an image, the 1-dimensional array ('vector') of M components in each element, and the 1-dimensional collection of T values (depending on the element type) making up each component. For each of the M components, there is an associated component name which can be read using dmGetCptName.

Context 1 (R1): Returns M = 1 since vector descriptors are not supported in R1.

Context 2 (R1.7): Returns M = the number of vector components; works for column, image data, coord, filter and image axis descriptors.

4.7.13 dmGetElementType

dmElementType dmGetElementType(dmDescriptor* dd)
char* dmGetElementTypeName(dmDescriptor* dd)

Return the element type of the descriptor.

Context 1 (R1): Initially returns always dmVALUE, the only element type supported at (R1).

Context 2 (R2.1): Return the appropriate element type for a column or key descriptor, or for a filter or coord descriptor. Most filters will have element type dmRANGE.

4.7.14 dmGetError

int dmGetError(void)

(R1): Retrieve the DataModel error status value. This routine can be called after any other call to check for success/failure. Return values will range from the general dmSUCCESS and dmFAILURE values to a number of routine-specific values like dmNOMOREROWS, dmNOMEM, etc.

4.7.15 dmGetErrorMessage

void dmGetErrorMessage(char* errMessage, int maxlen)

(R1): Like dmGetError, but returning a text string instead of numeric value.

4.7.16 dmGetIntervalType

dmIntervalType dmGetIntervalType(dmDescriptor* dd)
char* dmGetIntervalTypeName(dmDescriptor* dd)

(R4): Find the interval type of the descriptor. The interval type denotes whether an element of type RANGE or INTERVAL is to be considered a closed interval or an open interval. This is not yet implemented; the default is to assume closed intervals.

4.7.17 dmGetName

int dmGetName(dmDescriptor* dd, char *name, int maxlen)

(R1) Given a descriptor pointer, copy at most maxlen characters of the descriptor name into the pre-allocated char* name array.

4.7.18 dmGetScalar: read scalar value

short dmGetScalar_s(dmDescriptor* dd)
long dmGetScalar_l(dmDescriptor* dd)
float dmGetScalar_f(dmDescriptor* dd)
double dmGetScalar_d(dmDescriptor* dd)
void dmGetScalar_c(dmDescriptor* dd, char *value, int maxlen)
void dmGetScalar_br(dmDescriptor* dd, char* value, int maxlen)
int dmGetScalar_q(dmDescriptor* dd)
unsigned char dmGetScalar_ub(dmDescriptor* dd)
unsigned short dmGetScalar_us(dmDescriptor* dd)
unsigned long dmGetScalar_ul(dmDescriptor* dd)

Return the current value of a descriptor and force it to the desired type. The suffixes _q and _br denote logical and block reference types, respectively.

Context 1 (R1): Return the value of a scalar key of the same type as the routine.

Context 2 (R1): Return the value of a scalar key of a different type, casting the result to the type of the routine.

Context 3 (R1): Return the value of a scalar column descriptor, i.e. the cell value for this column and the current row, when the descriptor has the same data type as the routine.

Context 4 (R1): Return the value of a scalar column descriptor when it has a different data type, casting the result to the type of the routine.

Context 5 (R1.8): Return the value of a coord descriptor, by finding the value of its parent (e.g. dmGetScalar on the parent column) and applying its transformation to get the coord value, casting type if needed.

Context 6 (R2.1): Return the value of a column or key or subspace descriptor of compound element type, by applying the appropriate rules (return VALUE field for dmRANGE, (MAX+MIN)/2 for dmINTERVAL, etc.

Context 7 (R3): Return the first element of an array descriptor (including image data) and the first component of a vector descriptor. Not recommended.

Context 8 (R1.6): Make all the above happen correctly when the block is a virtual one, i.e. implement filtering.

4.7.19 dmGetScalars: read cells from multiple table rows

```
long dmGetScalars_s(dmDescriptor* dd, short *vals, long firstRow,long nrows)
long dmGetScalars_l(dmDescriptor* dd, long *vals, long firstRow,long nrows)
long dmGetScalars_f(dmDescriptor* dd, float *vals, long firstRow,long nrows)
long dmGetScalars_d(dmDescriptor* dd, double *vals,long firstRow,long nrows)
long dmGetScalars_q(dmDescriptor* dd,int *vals, long firstRow,long nrows)
long dmGetScalars_c(dmDescriptor* dd, char** vals, int maxlen,long firstRow,long nrows)
long dmGetScalars_br(dmDescriptor* dd, char **vals,int maxlen,long firstRow,long nrows
)
long dmGetScalars_ub(dmDescriptor* dd, unsigned char* vals,int firstRow, int nrows)
long dmGetScalars_us(dmDescriptor* dd, unsigned short* vals, int firstRow, int nrows)
long dmGetScalars_ul(dmDescriptor* dd, unsigned long* vals, int firstRow, int nrows)
```

These routines offer performance improvements for large tables by reading/writing multiple rows at once. They also are convenient for small tables (where buffering efficiency is not relevant) allowing a column-oriented approach.

Context 1 (R1): Read multiple rows of a scalar column. The first row number is given explicitly. The array of data is written to the storage allocated in the user program. Only works for table columns. Results are undefined if the call type does not match the column datatype. Note that for string and blockref columns care must be taken to ensure maxlen value matches the string length of the column being read. The **dmGetStrLen** function may be used to determine this value if it is not known ahead of time.

Context 2 (R1.6): Make all the above happen correctly when the block is a virtual one, i.e. implement filtering.

4.7.20 dmGetStrLen

```
long dmGetStrLen(dmDescriptor* dd)
```

Context 1 (R1.5): Get the string length of a string-valued table column.
Context 2 (R1.5): Get the length of a string keyword value.
Context 3 (R1.5): When called for any other descriptor, returns 0.

4.7.21 dmGetUnit

int dmGetUnit(dmDescriptor* dd, char *unit, int maxlen)

Context 1 (R1): Get the unit of a column descriptor.
Context 2 (R1.5): Get the unit of a key descriptor.
Context 3 (R1.8): Get the unit of other descriptors.

4.7.22 dmGetVector: Get vector value

dmGetVector_s(dmDescriptor* dd, short* result, long dim)
long* dmGetVector_l(dmDescriptor* dd, long* result, long dim)
float* dmGetVector_f(dmDescriptor* dd, float* result, long dim)
double* dmGetVector_d(dmDescriptor* dd, double* result, long dim)
int* dmGetVector_q(dmDescriptor* dd, int* result, long dim)
char** dmGetVector_c(dmDescriptor* dd, char** result, long dim)
char** dmGetVector_br(dmDescriptor* dd, char** result, long dim)
unsigned char* dmGetVector_ub(dmDescriptor* dd, unsigned char* result, long dim)
unsigned short* dmGetVector_us(dmDescriptor* dd, unsigned short* result, long dim)
unsigned long* dmGetVector_ul(dmDescriptor* dd, unsigned long* result, long dim)

Context 1 (R1.9): Get the value of a vector column data descriptor given its handle. The 'dim' argument is an input argument which gives the maximum number of values to return. Spec for this routine has been changed to assume preallocated memory (JCM 98/03/29).
Context 2 (R1.9): Cast type if necessary (dd data type and routine type in disagreement).
Context 3 (R1.9): Work correctly for scalar case (returns one value, effect is same as dmGetScalar).
Context 4 (R1.9): Return value for vectored coordinate descriptor, by calling dmGetVector for the parent descriptor and then applying the coord transform.
Context 5 (R2): Work for vectored header keys.

4.7.23 dmGetVectors: read cells from multiple table rows

long dmGetVectors_s(dmDescriptor* dd, short *vecs, int firstRow, int nrows)
long dmGetVectors_l(dmDescriptor* dd, long* vecs, int firstRow, int nrows)
long dmGetVectors_f(dmDescriptor* dd, float* vecs,, int firstRow, int nrows)
long dmGetVectors_d(dmDescriptor* dd, double *vecs, int firstRow, int nrows)
long dmGetVectors_q(dmDescriptor* dd, int *vecs, int firstRow, int nrows)
long dmGetVectors_c(dmDescriptor* dd, char **vecs, int firstRow, int nrows)
long dmGetVectors_br(dmDescriptor* dd, char *vecs, int firstRow, int nrows)
long dmGetVectors_ub(dmDescriptor* dd, unsigned char *vecs, int firstRow, int nrows)
long dmGetVectors_us(dmDescriptor* dd, unsigned short *vecs,int firstRow, int nrows)
long dmGetVectors_ul(dmDescriptor* dd, unsigned long* vecs, int firstRow, int nrows)

Context 1 (R1.9): Read multiple rows of a vector column. The first row number is given explicitly. The array of data is written to storage allocated in the user program. The number of values returned is the number of rows times the element dimension of the column. Only works for table columns.

4.7.24 dmGetVersion

int dmGetVersion(char *versionString, int maxlen)

(R1): Returns a string indicating the current release of the ASC interface.

4.8 dmImage Routines

4.8.1 dmImageClose

dmImageClose(dmBlock* image)

(R1): Close image block and associated dataset. Used for blocks opened by dmImageCreate and dmImageOpen. Identical with dmDatasetTableClose.

4.8.2 dmImageCreate

dmBlock* dmImageCreate(char* vdataSpec, dmDataType type, long* axes, long naxes)

(R1): Create an image of specified data type and dimensions, in a new dataset. This routine sets up the structure, it does not necessarily allocate memory for the data. This call specifies creating an image which will be stored as binned data in the usual IRAF IMH/FITS IMAGE way. This routine creates an image where each axis group is of dimensionality 1 and has name AXISn.

To create an image in an existing dataset, use the dmDatasetCreateImage routine.

4.8.3 dmImageDataGetPixel

short* dmImageDataGetPixel_s(dmDescriptor* data, long* pixelNo)
long* dmImageDataGetPixel_l(dmDescriptor* data, long* pixelNo)
float* dmImageDataGetPixel_f(dmDescriptor* data, long* pixelNo)
double* dmImageDataGetPixel_d(dmDescriptor* data, long* pixelNo)
unsigned char* dmImageDataGetPixel_ub(dmDescriptor* data, long* pixelNo)
unsigned short* dmImageDataGetPixel_us(dmDescriptor* data, long* pixelNo)
unsigned long* dmImageDataGetPixel_ul(dmDescriptor* data, long* pixelNo)

Context 1 (R2.0): Get a single pixel in an image. Example for 3D image:

```
long pixelNo[3] =  { 512, 512, 4 };
dmDescriptor* data = dmImageGetDataDescriptor( image );
double value = dmImageDataGetPixel\_d( data, pixelNo );
```

Context 2 (R2.0): Same for array column in table.
Context 3 (R3): Force (cast) type of result if it doesn't match the descriptor's data type.

4.8.4 dmImageDataGetPixlist: Get pixel list

int dmImageDataGetPixlist_s(dmDescriptor* data, int** pixlist, short* data, int maxSize, int* npixels)
int dmImageDataGetPixlist_l(dmDescriptor* data, int** pixlist, long* data, int maxSize, int* npixels)
int dmImageDataGetPixlist_f(dmDescriptor* data, int** pixlist, float* data, int maxSize, int* npixels)
int dmImageDataGetPixlist_d(dmDescriptor* data, int** pixlist, double* data, int maxSize, int* npixels)
int dmImageDataGetPixlist_ub(dmDescriptor* data, int** pixlist, unsigned char* data, int maxSize, int* npixels)
int dmImageDataGetPixlist_us(dmDescriptor* data, int** pixlist, unsigned short* data, int maxSize, int* npixels)
int dmImageDataGetPixlist_ul(dmDescriptor* data, int** pixlist, unsigned long* data, int maxSize, int* npixels)

Context 1 (R3): Return the image data as a list of pixels and data values. Only nonzero values are returned. npixels contains the number of such nonzero values, which is truncated to be at most maxSize; data contains the values; and each entry in pixlist is an integer array of values representing the pixel.
Context 2 (R3): Same but for an array column in a table.

4.8.5 dmImageDataGetPixlistSize

long dmImageDataGetPixlistSize(dmDescriptor* data)

Context 1 (R3): Return the number of nonzero pixels in the image.
Context 2 (R3): Same, for an array column in a table.

4.8.6 dmImageDataGetSubArray

int dmImageDataGetSubArray_s(dmDescriptor* data, long* pixelLl, long* pixelUr, short *values)

int dmImageDataGetSubArray_l(dmDescriptor* data, long* pixelLl, long* pixelUr, long *values)
int dmImageDataGetSubArray_f(dmDescriptor* data, long* pixelLl, long* pixelUr,float *values)
int dmImageDataGetSubArray_d(dmDescriptor* data, long* pixelLl, long* pixelUr,double *values)
int dmImageDataGetSubArray_ub(dmDescriptor* data, long* pixelLl, long* pixelUr,unsigned char *values)
int dmImageDataGetSubArray_us(dmDescriptor* data, long* pixelLl, long* pixelUr, unsigned short *values)
int dmImageDataGetSubArray_ul(dmDescriptor* data, long* pixelLl, long* pixelUr, unsigned long* values)

Context 1 (R1): Retrieve the values of a rectangular image subsection, specifying the lower left pixel and the upper right pixel, and returning the values in standard 1-D array order. Recall that pixel (1,1) is in the lowest and left-most pixel in an image. For example, for a 3-dimensional image, one might do:

```
long lower_left[3] = { 4, 8, 2 };
long upper_right[3] = { 6, 8, 4 };
double values[9];
dmBlock* image = dmImageOpen( "image.dat" );
dmDescriptor* data = dmImageGetDataDescriptor( image );
dmImageDataGetSubArray_l( data, lower_left, upper_right, values );
```

Then the array `values` will contain the following 9 values in this order: data(4,8,2), data(5,8,2), data(6,8,2), data(4,8,3), data(5,8,3), data(6,8,3), data(4,8,4), data(5,8,4), data(6,8,4).

Context 2 (R3): Same when descriptor is an array column instead of an image block's data descriptor. Example:

```
long lower_left[3] = { 4, 8, 2 };
long upper_right[3] = { 6, 8, 4 };
double values[9];
dmBlock* table = dmDatasetTableOpen( "table.dat[EVENTS]" );
dmDescriptor* data = dmTableOpenColumn( table, "IMAGE" );
dmImageDataGetSubArray_l( data, lower_left, upper_right, values );
```

Context 3 (R1.8): Make all the above happen correctly when the block is a virtual one, i.e. implement filtering.

4.8.7 dmImageDataInterpolate: Interpolate in image

short dmImageDataInterpolate_s(dmDescriptor* data, double* coordVal)
long dmImageDataInterpolate_l(dmDescriptor* data, double* coordVal)
float dmImageDataInterpolate_f(dmDescriptor* data, double* coordVal)
double dmImageDataInterpolate_d(dmDescriptor* data, double* coordVal)
unsigned char dmImageDataInterpolate_ub(dmDescriptor* data, double* coordVal)
unsigned short dmImageDataInterpolate_us(dmDescriptor* data,double* coordVal)
unsigned long dmImageDataInterpolate_ul(dmDescriptor* data,double* coordVal)

(R3): Linear interpolation on nearest neighbour pixels, returning interpolated pixel value. Example:

```
double coordVal[2] = { 4.5, 2.0 };
float result = dmImageDataInterpolate_f( data, coordVal );
```

The result will be 0.5 * (data(4,2) + data(5,2)).

4.8.8 dmImageDataSetPixel

int dmImageDataSetPixel_s(dmDescriptor* data, long* pixelNo, short pixelValue)
int dmImageDataSetPixel_l(dmDescriptor* data, long* pixelNo, long pixelValue)
int dmImageDataSetPixel_f(dmDescriptor* data, long* pixelNo, float pixelValue)
int dmImageDataSetPixel_d(dmDescriptor* data, long* pixelNo, double pixelValue)
int dmImageDataSetPixel_ub(dmDescriptor* data, long* pixelNo, unsigned char pixelValue)
int dmImageDataSetPixel_us(dmDescriptor* data, long* pixelNo, unsigned short pixelValue)
int dmImageDataSetPixel_ul(dmDescriptor* data, long* pixelNo, unsigned long pixelValue)

Context 1 (R2): Return or set a single image pixel value in an n-dimensional image data array. The pixelNo argument is an array of naxes integers; it is up to the caller to give the correct number of values. Example:

```
long pixelNo[3] =  { 512, 512, 4 };
double value = 14.8;
dmDescriptor* data = dmImageGetDataDescriptor( image );
dmImageDataSetPixel\_d( data, pixelNo, value );
```

which alters a single pixel value in the array.

Context 2 (R2): Same for an array column in a table.

4.8.9 dmImageDataSetPixlist

int dmImageDataSetPixlist_s(dmDescriptor* imdata, int** pixlist, short* data, int maxSize, int* npixels)
int dmImageDataSetPixlist_l(dmDescriptor* imdata, int** pixlist, long* data, int maxSize, int* npixels)
int dmImageDataSetPixlist_f(dmDescriptor* imdata, int** pixlist, float* data, int maxSize, int* npixels)
int dmImageDataSetPixlist_d(dmDescriptor* imdata, int** pixlist, double* data, int maxSize, int* npixels)
int dmImageDataSetPixlist_ub(dmDescriptor* imdata, int** pixlist, unsigned char* data, int maxSize, int* npixels)
int dmImageDataSetPixlist_us(dmDescriptor* imdata, int** pixlist, unsigned short* data, int maxSize, int* npixels)
int dmImageDataSetPixlist_ul(dmDescriptor* imdata, int** pixlist, unsigned long* data, int maxSize, int* npixels)

Context 1 (R3): Return or set the image pixel values as a list of pixel numbers and their values, only for pixels whose value is nonzero. For instance, for a 2D image the data might be

```
pixel_list[0] = (1,1)       data[0] = 38
pixel_list[1] = (29,13)     data[1] = 42
```

ASC Images will use the Fortran/IRAF/FITS convention that the lower left pixel number is (1,1), NOT the C count-from-zero convention. That doesn't affect how the C array pointers are returned, it just means that in C you access the elements as image[ypix-1][xpix-1] (etc).

The maxSize argument is an input argument to declare the actual size of the arrays being passed.

Context 2 (R3): Same but for an array column in a table.

4.8.10 dmImageDataSetSubArray

int dmImageDataSetSubArray_s(dmDescriptor* data, long* pixelLl, long* pixelUr, short* values)
int dmImageDataSetSubArray_l(dmDescriptor* data, long* pixelLl, long* pixelUr, long* values)
int dmImageDataSetSubArray_f(dmDescriptor* data, long* pixelLl, long* pixelUr, float* values)
int dmImageDataSetSubArray_d(dmDescriptor* data, long* pixelLl, long* pixelUr, double* values)
int dmImageDataSetSubArray_ub(dmDescriptor* data, long* pixelLl, long* pixelUr, char* values)

int dmImageDataSetSubArray_us(dmDescriptor* data, long* pixelLl, long* pixelUr, unsigned short* values)
int dmImageDataSetSubArray_ul(dmDescriptor* data, long* pixelLl, long* pixelUr, unsigned long* values)

Context 1 (R1): Set a rectangular subarray in an image. See dmImageGetSubArray for details.
Context 2 (R2): Same for array column in a table.

4.8.11 dmImageGetDataDescriptor

dmDescriptor* dmImageGetDataDescriptor(dmBlock* image)

Context 1 (R1): Return the data descriptor for the single cell in the image table. This cell contains the data. The data itself can be retrieved using the getArray calls or set using the setArray calls. It is a little extra work for the user to have a separate handle for the image metadata (the block handle) and the image data (the descriptor handle) but it will later allow us to use images in table cells exchangeably with images in image datablocks.

Context 2 (R2): If the block is actually a table, check to see if the table has only one column and that column has array dimension greater than zero. In that case, return the column descriptor. Otherwise, return null and set an error condition.

4.8.12 dmImageOpen

dmBlock* dmImageOpen(char* vdataSpec)

Context 1 (R1): Opens an image via a virtual data specification (which will normally refer to a dataset containing a single image). See the dmDatasetTableOpen call and/or the the DataModel filtering API for more details on the syntax of the virtual data specification.

Context 2 (R1.5): If no image name is specified within vdataSpec, the first block in the specified dataset is opened.

Context 3 (R1.7; partially implemented for R1.5): Open a filtered (array subsection) image and dataset.

Context 4 (R3): Open an array in a table cell as if it were an image. The virtual spec is of the form "dataset[table][rowno]colname". The virtual datablock has the header and datasubspace of the table, and the image data descriptor is the column data descriptor. Other image operations are as usual. The dmTableNextRow command may NOT be used to alter the row since we have opened a single image, not the whole table.

4.9 dmKernel Routines

4.9.1 dmKernelGetCopy: Get copy kernel

int dmKernelGetCopy(char *copyKernelName, int maxlen)

(R1): Return the current copy kernel name.

4.9.2 dmKernelGetCreate: Get creation kernel

int dmKernelGetCreate(char *creationKernelName, int maxlen)

(R1): Return the current creation kernel name.

4.9.3 dmKernelGetList

int dmKernelGetList(char ***kernelIDs, int* numKernels)

(R1.6): Provides a list of the ETOOLs kernels available for current run-time use. Unlike most other string routines, the kernelIDs array must be freed by the caller after use.

4.9.4 dmKernelSetCopy: Set copy kernel

char* dmKernelSetCopy(char* kernelId)

(R1): Specify the kernel to associate with an opened Virtual Table being copied. The default behaviour is for the copy to retain the kernel of the underlying input dataset, but this routine allows you to override. Calling it with a blank or null argument resets to the default behaviour.

4.9.5 dmKernelSetCreate: Set creation kernel

char* dmKernelSetCreate(char* kernelId)

(R1): Specify the kernel to be used for creating new datasets, returning the kernel actually set, or null on error. Currently supported values are:

Kernel Mnemonic	Description
dmFITSKERNEL	(interface to CFITSIO)
dmIRAFKERNEL	(interface to native IRAF)
blank or null	reset to default (FITS)

4.10 dmKey Routines

4.10.1 dmKeyCreate

dmDescriptor* dmKeyCreate(dmBlock* block, char* name, dmDataType type, char* unit, char* description)

(R2.1): Create a single, scalar header keyword. This routine creates a key descriptor. You can then set its values using the dmSetScalar, etc., routines. Most often, you'll instead probably want the dmKeyWrite set of routines which write the values at the same time, since unlike columns keys only have a single value.

4.10.2 dmKeyCreateGeneric

dmDescriptor* dmKeyCreateGeneric(dmBlock* block, char* name, dmDataType type, char* unit, char* desc, dmElementType* elementType, char** cptNames, long dim, long size)

(R2.1): Create a generic header keyword. Support vectored and compound element type keys. This creates a descriptor; you need to assign the values as well, using the same routines as for table columns.

4.10.3 dmKeyGetNo

long dmKeyGetNo(dmDescriptor* dd)

Context 1 (R1.5): Return the number of the key in the header, given its descriptor.
Context 2 (R1.5): Return zero if descriptor is not a key.

4.10.4 dmKeyOpen: Search by name for descriptor

dmDescriptor* dmKeyOpen(dmBlock* block, char* name)

Context 1 (R1): Searches for a key (or column) by name, returning a descriptor, after which the dmGetScalar, dmGetArray, or dmGetVector calls would be used to retrieve data values. By the Greenbank convention, column and header entries are interchangeable so either keys or columns may be found, but header keys will returned first.
In (R1.5) the search will be case insensitive. Accessing a header DD as if it were a column makes it look like a column, all of whose rows have equal values; accessing a column DD as if it were a header gives you the value in the current row of the table. However, the dmKeyGetNo or dmColumnGetNo routines will return zero in these cases.

Context 2 (R1.9): If no match on a key or column name, look for a match on a key or column component name. For instance, if searching for DETY and a vector column DETPOS=(DETX,DETY) exists, and there is no key or column called DETY, make a new descriptor for DETY as if it were a scalar.

4.10.5 dmKeyRead: Read header scalar attribute

dmDescriptor* dmKeyRead_s(dmBlock* block, char* name, short* value)
dmDescriptor* dmKeyRead_l(dmBlock* block, char* name, long* value)
dmDescriptor* dmKeyRead_f(dmBlock* block, char* name, float* value)
dmDescriptor* dmKeyRead_d(dmBlock* block, char* name, double* value)
dmDescriptor* dmKeyRead_q(dmBlock* block, char* name, int* value)
dmDescriptor* dmKeyRead_c(dmBlock* block, char* name, char* value, int maxlen)
dmDescriptor* dmKeyRead_br(dmBlock* block, char* name, char* value, int maxlen)
dmDescriptor* dmKeyRead_ub(dmBlock* block, char* name, unsigned char* value)
dmDescriptor* dmKeyRead_us(dmBlock* block, char* name, unsigned short* value)
dmDescriptor* dmKeyRead_ul(dmBlock* block, char* name, unsigned long* value)

Context 1 (R1): Search for a header key by name. The first key to give a case-insensitive match is found. Return its descriptor and its value. The value is forced to the desired type if necessary, and truncated to length maxlen in the case of string types. Return a null descriptor if the key is not present in the header (no error condition is set).

Context 2 (R2.1): Case of a vector or array key. Return the value of the first component of the first element. (The user may check the descriptor to find out the array dimension etc).

Context 3 (R3): Case of a column. Search also for a match with a column name. Return the column's descriptor and the value for the current row. The intent is that the user doesn't care if the value is a column or a key.

4.10.6 dmKeyReadVector: Read header vector keyword

dmDescriptor* dmKeyReadVector_s(dmBlock* block, char* name, short* value, long dim, long* nvals)
dmDescriptor* dmKeyReadVector_l(dmBlock* block, char* name, long* value, long dim, long* nvals)
dmDescriptor* dmKeyReadVector_f(dmBlock* block, char* name, float* value, long dim, long* nvals)
dmDescriptor* dmKeyReadVector_d(dmBlock* block, char* name, double* value, long dim, long* nvals)
dmDescriptor* dmKeyReadVector_c(dmBlock* block, char* name, char* value, long dim, int maxlen, long* nvals)
dmDescriptor* dmKeyReadVector_br(dmBlock* block, char* name, char* value, long dim, int maxlen, long* nvals)
dmDescriptor* dmKeyReadVector_q(dmBlock* block, char* name, int* value, long dim,

long* nvals)
dmDescriptor* dmKeyReadVector_ub(dmBlock* block, char* name, unsigned char* value, long dim, long* nvals)
dmDescriptor* dmKeyReadVector_us(dmBlock* block, char* name, unsigned short* value, long dim, long* nvals)
dmDescriptor* dmKeyReadVector_ul(dmBlock* block, char* name, unsigned long* value, long dim, long* nvals)

Context 1 (R2.1): Read at most dim values from a vectored array header key, given its name. Return descriptor and values, and (in nvals) number of values actually read. The space for the value array must be allocated by the user. The descriptor can be used to find element dimension, array dimension, and component names using dmGetDim, dmGetArrDim, dmGetCptName, etc.

Context 2 (R2.1): Case of vectored key which is not an array.

Context 3 (R2.1): Case of array key which is not vectored.

Context 4 (R2.1): Case of scalar key, returns with the appropriate value and nvals = 1.

Context 5 (R3): Case of a column. Search also for a match with a column name. Return the column's descriptor and the value for the current row. The intent is that the user doesn't care if the value is a column or a key.

4.10.7 dmKeyWrite: Write scalar header key

dmDescriptor* dmKeyWrite_s(dmBlock* block, char* name, short value, char* unit, char* desc)
dmDescriptor* dmKeyWrite_l(dmBlock* block, char* name, long value, char* unit, char* desc)
dmDescriptor* dmKeyWrite_f(dmBlock* block, char* name, float value, char* unit, char* desc)
dmDescriptor* dmKeyWrite_d(dmBlock* block, char* name, double value, char* unit, char* desc)
dmDescriptor* dmKeyWrite_q(dmBlock* block, char* name, int value, char* unit, char* desc)
dmDescriptor* dmKeyWrite_c(dmBlock* block, char* name, char* value, char* unit, char* desc)
dmDescriptor* dmKeyWrite_br(dmBlock* block, char* name, char* value, char* unit, char* desc)
dmDescriptor* dmKeyWrite_ub(dmBlock* block, char* name, unsigned char value, char* unit, char* desc)
dmDescriptor* dmKeyWrite_us(dmBlock* block, char* name, unsigned short value, char* unit, char* desc)
dmDescriptor* dmKeyWrite_ul(dmBlock* block, char* name, unsigned long value, char* unit, char* desc)

(R1): Write a single, scalar header keyword of the specified type. Note that attributes are required to be uniquely named, so a second write call using the same name will overwrite the previous value. With this exception, the keyword is written at the 'current header position' (usually the end of the header). The unit and description for the keyword (fully supported in R1.5) must also be supplied (but may be null or blank).

4.10.8 dmKeyWriteArray: Write array header key

dmDescriptor* dmKeyWriteArray_s(dmBlock* block, char* name, short* value, char* unit, char* desc, long size)
dmDescriptor* dmKeyWriteArray_l(dmBlock* block, char* name, long* value, char* unit, char* desc, long size)
dmDescriptor* dmKeyWriteArray_f(dmBlock* block, char* name, float* value, char* unit, char* desc, long size)
dmDescriptor* dmKeyWriteArray_d(dmBlock* block, char* name, double* value, char* unit, char* desc, long size)
dmDescriptor* dmKeyWriteArray_q(dmBlock* block, char* name, int* value, char* unit, char* desc, long size)
dmDescriptor* dmKeyWriteArray_c(dmBlock* block, char* name, char** value, char* unit, char* desc, long size)
dmDescriptor* dmKeyWriteArray_ub(dmBlock* block, char* name, unsigned char* value, char* unit, char* desc, long size)
dmDescriptor* dmKeyWriteArray_us(dmBlock* block, char* name, unsigned short* value, char* unit, char* desc, long size)
dmDescriptor* dmKeyWriteArray_ul(dmBlock* block, char* name, unsigned long* value, char* unit, char* desc, long size)

(R2.1): Write a 1-D array header keyword with the specified size and values. Arrayed keys should be read using the general descriptor function dmGetArray.

```
Note: We currently allow users to have scalar keys and arrayed
keys which have the same alphabetic prefix.  Therefore if one
has an array called 'FOO', one may write a separate scalar
keyword of the form 'FOO#', which would be interpreted as an
element of the array.  This feature can cause problems if not
used carefully.
```

4.10.9 dmKeyWriteElement: Write Element header key

dmDescriptor* dmKeyWriteElement_s(dmBlock* block, char* name, short** value, char* unit, char* desc, char** cptNames, char* elementType)

dmDescriptor* dmKeyWriteElement_l(dmBlock* block, char* name, long** value, char* unit, char* desc, char** cptNames, char* elementType)
dmDescriptor* dmKeyWriteElement_f(dmBlock* block, char* name, float** value, char* unit, char* desc, char** cptNames, char* elementType)
dmDescriptor* dmKeyWriteElement_d(dmBlock* block, char* name, double** value, char* unit, char* desc, char** cptNames, char* elementType)

Context 1 (R2.1): Write an interval or value/uncertainty set to the header. Specify the element type and its component names. *Note change to argument list 1997 Jul 21; routine renamed 1998 Mar 29.*

Context 2 (R2.1): If the component name array is null, generate the component names automatically from the element type using some default rules.

4.10.10 dmKeyWriteVector: Write vectored header key

dmDescriptor* dmKeyWriteVector_s(dmBlock* block, char* name, short* value, char* unit, char* desc, char** cptNames, long dim)
dmDescriptor* dmKeyWriteVector_l(dmBlock* block, char* name, long* value, char* unit, char* desc, char** cptNames, long dim)
dmDescriptor* dmKeyWriteVector_f(dmBlock* block, char* name, float* value, char* unit, char* desc, char** cptNames, long dim)
dmDescriptor* dmKeyWriteVector_d(dmBlock* block, char* name, double* value, char* unit, char* desc, char** cptNames, long dim)
dmDescriptor* dmKeyWriteVector_q(dmBlock* block, char* name, int* value, char* unit, char* desc, char** cptNames, long dim)
dmDescriptor* dmKeyWriteVector_c(dmBlock* block, char* name, char** value, char* unit, char* desc, char** cptNames, long dim)
dmDescriptor* dmKeyWriteVector_ub(dmBlock* block, char* name, unsigned char* value, char* unit, char* desc, char** cptNames, long dim)
dmDescriptor* dmKeyWriteVector_us(dmBlock* block, char* name, unsigned short* value, char* unit, char* desc, char** cptNames, long dim)
dmDescriptor* dmKeyWriteVector_ul(dmBlock* block, char* name, unsigned long* value, char* unit, char* desc, char** cptNames, long dim)

(R2.1): Write a vector header keyword with the specified size, component names and values.

4.11 dmSet Routines

4.11.1 dmSetArray

int dmSetArray_s(dmDescriptor* dd, short* value, long npixels)
int dmSetArray_l(dmDescriptor* dd, long* value, long npixels)

int dmSetArray_f(dmDescriptor* dd, float* value, long npixels)
int dmSetArray_d(dmDescriptor* dd, double* value, long npixels)
int dmSetArray_q(dmDescriptor* dd, int* value, long npixels)
int dmSetArray_c(dmDescriptor* dd, char** value, long npixels)
int dmSetArray_br(dmDescriptor* dd, char** value, long npixels)
int dmSetArray_ub(dmDescriptor* dd, unsigned char* value, long npixels)
int dmSetArray_us(dmDescriptor* dd, unsigned short* value, long npixels)
int dmSetArray_ul(dmDescriptor* dd, unsigned long* value, long npixels)

Context 1 (R1): Set value of an array column data descriptor. Input is a 1-D array of values, ignoring the fact that the array may actually be a higher- dimensional object. The pixel values are given in FORTRAN-like order, ie, 1-based (not 0-based). If the descriptor is a variable-length array (supported in R1.8), npixels is assumed to be the length for the current row.

Context 2 (R1): Give an error if the data type of the descriptor does not match that of the routine.

Context 3 (R1): Write data to an image block (set value of image data descriptor).

Context 4 (R2.1): Write value of an array key descriptor.

4.11.2 dmSetCptName

int dmSetCptName(dmDescriptor* dd, long cptNo, char* name)

Context 1 (R1.8): Change the name of the single component of a scalar descriptor (element dim = 1). Since the component name and the descriptor name are required to be the same in this case, the effect is the same as dmSetName.

Context 2 (R1.8): Set or change the name of one component of a vector descriptor. The descriptor must have element dimension greater than or equal to cptNo.

4.11.3 dmSetDesc

int dmSetDesc(dmDescriptor* dd, char* desc)

(R1.5) The descriptor Description is a text string which may include spaces and any ASCII text. It is recommended that the string be 40 bytes or less. The intent is to provide an explanatory label to accompany the concise, spaceless descriptor name.

4.11.4 dmSetDisp

int dmSetDisp(dmDescriptor* dd, char* dispFormat)

(R1.5): Set the display format hint for a descriptor. The display format hint is an abstraction of the FITS TDISPn mechanism, and is provided to help browser programs find a suitable format for displaying the data. In particular, your browser program might guess that even doubles don't usually need more than 10 digits of precision to inspect their values, and this is usually true, but not in the case of TIME values which often need more precision. So rather than have lots of extra blank space everywhere to cover the worst case, we'll assume a typical case and flag the nasty ones with a dispFormat hint. For compatibility with FITS, the value of TDISPn (and dispFormat) is a Fortran 90 compatible format string. The default value of dispFormat is a blank string.

4.11.5 dmSetElement

int dmSetElement_s(dmDescriptor* dd, short* value, dmElementType type)
int dmSetElement_l(dmDescriptor* dd, long* value, dmElementType type)
int dmSetElement_f(dmDescriptor* dd, float* value,dmElementType type)
int dmSetElement_d(dmDescriptor* dd, double* value, dmElementType type)

Set the value of a data descriptor of arbitrary element type.

Context 1 (R2.1): Set compound value of a column data descriptor when the descriptor has the same element type as the input argument element type. Assumes an input array of 0, 1, 2 or 3 values depending on the element type.

Context 2 (R2.1): When descriptor has a different element type from input element type, make the necessary conversion.

Context 3 (R2.1): Support key descriptors.

Context 4 (R2.1): Support subspace descriptors.

4.11.6 dmSetKernelOption

int dmSetKernelOption(char* option)

(R2): Set an internal kernel parameter. This mechanism will be used to tune particular kernel parameters. At the moment only the TABLE options are supported:

Option	Effect
'TABLE=BINTABLE'	Write tables in BINTABLE format if kernel is FITS
'TABLE=ASCII'	Write tables in ASCII table format if kernel is FITS
'TABLE=QPEVT'	Write tables in QPOE event format if kernel is IRAF
'TABLE=STSDAS'	Write tables in STSDAS format if kernel is IRAF

If you call with a kernel option that has no meaning for the current kernel, it is simply ignored.

4.11.7 dmSetLimit: Set upper limit

int dmSetLimit_s(dmDescriptor* dd, short* value)
int dmSetLimit_l(dmDescriptor* dd, long* value)

int dmSetLimit_f(dmDescriptor* dd, float* value)
int dmSetLimit_d(dmDescriptor* dd, double* value)

Context 1 (R3): Write an upper limit: Set the value of an interval data descriptor as an upper limit. Element type dmVALUE: just use the value, but return an error code.

Context 2 (R3): Element type dmRANGE or dmINTERVAL: set MAX and VALUE to value, and MIN to zero.

4.11.8 dmSetName

int dmSetName(dmDescriptor* dd, char* name)

(R2) Set or change the name of the descriptor, irrespective of descriptor type.

4.11.9 dmSetScalar: Set scalar value

int dmSetScalar_s(dmDescriptor* dd, short value)
int dmSetScalar_l(dmDescriptor* dd, long value)
int dmSetScalar_f(dmDescriptor* dd, float value)
int dmSetScalar_d(dmDescriptor* dd, double value)
int dmSetScalar_q(dmDescriptor* dd, int value)
int dmSetScalar_c(dmDescriptor* dd, char* value)
int dmSetScalar_br(dmDescriptor* dd, char value)
int dmSetScalar_ub(dmDescriptor* dd, unsigned char value)
int dmSetScalar_us(dmDescriptor* dd, unsigned short value)
int dmSetScalar_ul(dmDescriptor* dd, unsigned long value)

Set the value of a data descriptor (in a header or the current row of a column) given its handle.

Context 1 (R1): Set the value of a scalar key; it must have the same data type as the routine.

Context 2 (R1): Set the cell value for a scalar column in the current row.

Context 3 (R1.8): Setting the value of a coord descriptor involves applying the inverse coordinate transform and setting the value of its parent descriptor. For example, if EQPOS(RA,DEC) is a vector coord descriptor defined as a transformation on a parent column POS(X,Y), then setting the value of EQPOS actually has the effect of changing the value of POS. Support for inverting the transform is NOT guaranteed to be present for all transforms; the return value of the function is the non-zero value dmNO_COORD_INVERSE if an error of this kind occurs.

Context 4 (R2.1): Set the value of a descriptor which has compound element type by applying the appropriate rules (i.e. set VALUE = MIN = MAX).

Context 5 (R3): Set all the values of an array descriptor to the same value. (or, define SetScalar on an array descriptor to be an error; we should review the appropriate behaviour).

4.11.10 dmSetScalars: write cells to multiple table rows

long dmSetScalars_s(dmDescriptor* dd, short* value, int firstRow, int nrows)
long dmSetScalars_l(dmDescriptor* dd, long* value, int firstRow, int nrows)
long dmSetScalars_f(dmDescriptor* dd, float* value, int firstRow, int nrows)
long dmSetScalars_d(dmDescriptor* dd, double* value,int firstRow, int nrows)
long dmSetScalars_q(dmDescriptor* dd, int* value, int firstRow, int nrows)
long dmSetScalars_c(dmDescriptor* dd, char** value, int firstRow, int nrows)
long dmSetScalars_br(dmDescriptor* dd, char** value, int firstRow, int nrows)
long dmSetScalars_ub(dmDescriptor* dd, unsigned char* value, int firstRow, int nrows)
long dmSetScalars_us(dmDescriptor* dd, unsigned short* value, int firstRow, int nrows)
long dmSetScalars_ul(dmDescriptor* dd, unsigned long* value, int firstRow, int nrows)

These routines offer performance improvements for large tables by reading/writing multiple rows at once.

Context 1 (R1): Write multiple rows of a scalar column. The first row number is given explicitly. The array of data is passed in from the user program. Only works for table columns. Results are undefined if the call type does not match the column datatype.

4.11.11 dmSetUnit

int dmSetUnit(dmDescriptor* dd, char* unit)

Set the unit for the descriptor. See the HEASARC document by Ian George on recommended unit strings. In particular, note that the unit of the second is 's', not 'sec', and that kilo electron volt is spelled 'keV'. This call will work only on table columns in (R1).

4.11.12 dmSetVector: Set vector value

int dmSetVector_s(dmDescriptor* dd, short* value, long dim)
int dmSetVector_l(dmDescriptor* dd, long* value, long dim)
int dmSetVector_f(dmDescriptor* dd, float* value, long dim)
int dmSetVector_d(dmDescriptor* dd, double* value, long dim)
int dmSetVector_q(dmDescriptor* dd, int* value, long dim)
int dmSetVector_c(dmDescriptor* dd, char** value, long dim)
int dmSetVector_br(dmDescriptor* dd, char** value, long dim)
int dmSetVector_ub(dmDescriptor* dd, unsigned char* value, long dim)
int dmSetVector_us(dmDescriptor* dd, unsigned short* value, long dim)
int dmSetVector_ul(dmDescriptor* dd, unsigned long* value, long dim)

Set the value of a vector data descriptor given its handle.

Context 1 (R1.9): Set the value of a vector column data descriptor given its handle. The 'dim' argument is an input argument which gives the number of values provided. If this is more than the element dimension, the extra values are ignored. If it is less, the missing values are set to zero.

Context 2 (R1.9): Work correctly for scalar case (sets only one value).

Context 3 (R1.9): Set value for vectored coordinate descriptor, by applying the inverse transform and calling dmSetVector for the parent descriptor. See dmSetScalar.

Context 4 (R2.1): Work for vectored keys.

4.11.13 dmSetVectors: write cells to multiple table rows

long dmSetVectors_s(dmDescriptor* dd, short* value, long dim, int firstRow, int nrows)
long dmSetVectors_l(dmDescriptor* dd, long* value, long dim, int firstRow, int nrows)
long dmSetVectors_f(dmDescriptor* dd, float* value, long dim, int firstRow, int nrows)
long dmSetVectors_d(dmDescriptor* dd, double* value, long dim, int firstRow, int nrows)
long dmSetVectors_q(dmDescriptor* dd, int* value, long dim, int firstRow, int nrows)
long dmSetVectors_c(dmDescriptor* dd, char** value, long dim, int firstRow, int nrows)
long dmSetVectors_br(dmDescriptor* dd, char** value, long dim, int firstRow, int nrows)
long dmSetVectors_ub(dmDescriptor* dd, unsigned char* value, long dim, int firstRow, int nrows)
long dmSetVectors_us(dmDescriptor* dd, unsigned short* value, long dim, int firstRow, int nrows)
long dmSetVectors_ul(dmDescriptor* dd, unsigned long* value, long dim, int firstRow, int nrows)

Context 1 (R1.9): Write multiple rows for a vector column. Explicitly give the number of values passed in for each row. The total number of values passed in is dim * nrows.

4.12 dmSubspace Routines

4.12.1 dmSubspaceColCreate: Create range filter

dmDescriptor* dmSubspaceColCreate_s(dmBlock* block, char* name, char* unit, short* value1, short* value2, long nvalues)
dmDescriptor* dmSubspaceColCreate_l(dmBlock* block, char* name, char* unit, long* value1, long* value2, long nvalues)
dmDescriptor* dmSubspaceColCreate_f(dmBlock* block, char* name, char* unit, float* value1, float* value2, long nvalues)
dmDescriptor* dmSubspaceColCreate_d(dmBlock* block, char* name, char* unit, double* value1, double* value2, long nvalues)

Context 1 (R1.6): Create a descriptor of element type dmRANGE, array dimensionality 1, array size `nvalues`, and store it in the data subspace of the block, thus giving it a descriptor type of dmSUBSPACE. Supply an array of pairs of values to store in the array. The component names for the compound descriptor are generated automatically.

The NAME in the case of subspaces and regions is the name of the variable being filtered, not a name for the specific region or filter. They are added as new columns to the DSS table.

The interpretation of a subspace column is that it represents a constraint which each row of the table satisfies: it is a set of good intervals of the named quantity, so that each row of the table represents data for which the named quantity is within one of those intervals. It is the user's responsibility to check that this is true; no actual filtering is done by this routine.

Context 2 (R1.6): Recognize the special case of a subspace column called TIME. Store it in a kernel-dependent way (GTI table for FITS; deffilt for QPOE).

Context 3 (R2): Recognize the special case of a subspace column called PHA. Store it both as a filter and as a pair of keys for back compatibility.

4.12.2 dmSubspaceColCreateTable: Create range subspace column with value table

dmDescriptor* dmSubspaceColCreateTable_s(dmBlock* block, char* table_name, char* name, char** cptNames, char* unit, short* value1, short* value2, long nvalues)
dmDescriptor* dmSubspaceColCreateTable_l(dmBlock* block, char* table_name, char* name, char** cptNames, char* unit, long* value1, long* value2, long nvalues)
dmDescriptor* dmSubspaceColCreateTable_f(dmBlock* block, char* table_name, char** cptNames, char* name, char* unit, float* value1, float* value2, long nvalues)
dmDescriptor* dmSubspaceColCreateTable_d(dmBlock* block, char* table_name, char* name, char** cptNames, char* unit, double* value1, double* value2, long nvalues)

Context 1 (R1.6): Like the dmSubspaceColCreate routines, except that a separate table is created in the same dataset in which block exists to store the filter values. This is useful if one expects to have many ranges applied to the same variable, since "non-special" filter values are stored in keywords by default and therefore have a practical limit on how many filter ranges can be stored. The cptNames parameter is an array of strings with a size of two, where each string is the name to give to the first and second columns respectively within the value table. Note that each component of each subspace column has its own value table and not every component is required to have a value table, although all the components of a given subspace column which do have a value table must have the same column names. Also note that if one does a dmSubspaceColCreateTable, resets the current component, and then does a dmSubspaceColSet, these newly set values will not be placed in a value table unless dmSubspaceColSetTableName is called on the current component (table name should be unique, even from that of value tables of other components).

4.12.3 dmSubspaceColGet

dmSubspaceColGet_s(dmDescriptor* dd, short** value1, short** value2, long* nvalues)
dmSubspaceColGet_l(dmDescriptor* dd, int** value1, int** value2, long* nvalues)
dmSubspaceColGet_f(dmDescriptor* dd, float** value1, float** value2, long* nvalues)
dmSubspaceColGet_d(dmDescriptor* dd, double** value1, double** value2, long* nvalues)

Context 1 (R1.6): Retrieve filter values given filter descriptor.

Context 2 (R2.1): Retrieve values given a table column of arbitrary element type, forcing to element type dmRANGE.

4.12.4 dmSubspaceColIntersect

int dmSubspaceColIntersect(dmDescriptor* dd1, dmDescriptor* dd2, dmDescriptor* dd)

(R1.9) Intersect two DSS DD's and copy the result to the third. Typically the three DD's are in three separate files and refer to the same quantity (e.g. all are TIME in three different tables).

4.12.5 dmSubspaceColOpen

dmDescriptor* dmSubspaceColOpen(dmBlock* block, char* name)

(R1.6): Find a subspace column in the data subspace of a block, given its name. Analogous to dmTableOpenColumn and dmKeyOpen. (*new routine*).

4.12.6 dmSubspaceColRead

dmDescriptor* dmSubspaceColRead_s(dmBlock* block, char* name, short** value1, short** value2, long* nvalues)
dmDescriptor* dmSubspaceColRead_l(dmBlock* block, char* name, int** value1, int** value2, long* nvalues)
dmDescriptor* dmSubspaceColRead_f(dmBlock* block, char* name, float** value1, float** value2, long* nvalues)
dmDescriptor* dmSubspaceColRead_d(dmBlock* block, char* name, double** value1, double** value2, long* nvalues)

(R1.6): Retrieve filter values given filter descriptor. Combines dmSubspaceColOpen and dmSubspaceColGet, analogous to dmKeyRead.

4.12.7 dmSubspaceColSet

dmSubspaceColSet_s(dmDescriptor* dd, short* value1, short* value2, long nvalues)
dmSubspaceColSet_l(dmDescriptor* dd, int* value1, int* value2, long nvalues)
dmSubspaceColSet_f(dmDescriptor* dd, float* value1, float* value2, long nvalues)
dmSubspaceColSet_d(dmDescriptor* dd, double* value1, double* value2, long nvalues)

(R1.6) Set filter values given filter descriptor. If necessary, alter the size of the array.

4.12.8 dmSubspaceColSetTableName

int dmSubspaceColSetTableName(dmDescriptor* col, char* name)

(R1.6): Set the name of the value table for the given data subspace descriptor.

4.12.9 dmSubspaceColUpdate

dmSubspaceColUpdate_s(dmDescriptor* dd,short* value1,short* value2,long nvalues)
dmSubspaceColUpdate_l(dmDescriptor* dd,int* value1,int* value2,long nvalues)
dmSubspaceColUpdate_f(dmDescriptor* dd,float* value1,float* value2,long nvalues)
dmSubspaceColUpdate_d(dmDescriptor* dd,double* value1,double* value2,long nvalues)

(R1.6): Set subspace column values given subspace column descriptor. If necessary, alter the size of the array. Like dmSubspaceColSet routines, except that input values are intersected with existing values. A NULL intersection causes the current component to be removed from the data subspace.

4.12.10 dmSubspaceCreateRegion: Create region subspace

dmDescriptor* dmSubspaceCreateRegion(dmBlock* block, char* name, char* type, char* unit, char* region, char** cptNames, long dim)

(R1.9): Create a 2D region and add it to the DSS. Use the PROS/SAOIMAGE region syntax.

4.12.11 dmSubspaceGetRegion

char* dmSubspaceGetRegion(dmDescriptor* dd, char* region)

(R1.9): Retrieve the region string stored in a region type filter.

4.12.12 dmSubspaceSetRegion

int dmSubspaceSetRegion(dmDescriptor* filter, char* region)

(R1.9) Store a region string in a region filter. It is an error to specify a region for a descriptor with a vector of dimensionality other than 2.

4.13 dmTable Routines

4.13.1 dmTableAllocRow

void* dmTableAllocRow(dmBlock* table)

(R1.6): This routines facilitates row-based I/O on arbitrary tables, whose structure need not be known apriori. For example, rather than statically defining your row structures prior to compilation, this routine allows dynamic runtime definitions by allocating a row structure of the appropriate size and structure for the specified table. The resultant row structure may be used (via the returned pointer) for subsequent dmGetRow and dmPutRow calls.

This approach will be even more useful in the future (R1.7), when the API is expanded to include routines that will allow you to access the indvidual columns of this arbitrarily defined row structure.

4.13.2 dmTableClose: Close table

int dmTableClose(dmBlock* table)

(R1): Provides a simple means of closing a table and its parent dataset at with just one call, releasing all associated memory and closing the underlying physical file(s). As it provides essentially the same functionality as does dmImageClose, this routine will also work correctly when passed an image block pointer.
(R1.7) Renamed, was dmDatasetTableClose.

4.13.3 dmTableCopyRow

int dmTableCopyRow(dmBlock* table1, dmBlock* table2)

(R2): This routine is used in conjunction with the dmBlockCreateCopy routine. If table2 has been created as a copy of table1, it 'remembers' the columns from table 1 that its columns correspond to (some may have been deleted, some new ones may have been added). The values for the current row are copied from table 1 to table 2. You can't use the usual row-based I/O routines to do this in general unless you happen to know the structure of the table. This routine lets you work in the paradigm of 'make me a copy of whatever is in this table, and then add the following extra information'.

4.13.4 dmTableCreate: Create table and dataset

dmBlock* dmTableCreate(char* vdataSpec)

R1: Create a table datablock and a dataset at the same time. The vdataSpec will specify both the dataset name and the table name using the syntax "dataset[table]". This syntax is compatible with the filtering syntax. The dataset pointer is not directly returned, but may be accessed by the getDataset routine. After creation, the table has no rows and no columns. If you want to create a table in an existing dataset, use dmDatasetCreateTable instead.

R1.7: Renamed routine, previously called dmDatasetTableCreate

4.13.5 dmTableCreateColumns

dmDescriptor** dmTableCreateColumns(dmBlock* table, char** names, dmDataType* types, char** units, char** desc, long ncols)

(R2.1): Create a set of scalar columns all at once. A convenience function to save lots of calls to dmColumnCreate.

4.13.6 dmTableCreateGenericColumns

dmDescriptor** dmTableCreateGenericColumns(dmBlock* table, char** names, dmDataType* types, char** units, char** desc, dmElementType* elementTypes, char** cptNames, long* dim, long** axes, long *naxes, long ncols)

(R2.1): Create a set of generic table columns all at once. The cptNames string array argument refers to the components of ALL columns, and is thus scanned according to each columns specified dimensionality and element type.

4.13.7 dmTableGetNoCols

long dmTableGetNoCols(dmBlock* table)

Context 1 (R1): For a table, returns the number of columns in the table, or dmBADCOL on error.

Context 2 (R1): For an image, returns 1 (an image is a table with 1 column and 1 row).

4.13.8 dmTableGetNoRows

long dmTableGetNoRows(dmBlock* table)

Context 1 (R1): For a table, returns the absolute number of physical rows in the table, or dmBADROW on error.

Note that it is difficult (for performance reasons, mainly) to determine apriori the number of virtual rows in a virtual table (ie, a table opened with an arbitrary filter). Without actually applying the filter to an entire scan of the table, how else could one determine such information? So, accurately determining numrows prior to application table scanning implies that two entire traversals would be necessary for an application doing filtered I/O, one hidden scan within the DM library to determine numrows, the other scan in the main processing loop of the application code. Because of this performance constraint, the DM *does not* traverse filtered tables in an attempt to correctly indicate the number of rows that pass the filter; users should generally check for dmNOMOREROWS to terminate filtered table traversal loops, rather than looping on the absolute row number obtained from dmTableGetNoRows.

Context 2 (R1): For an image, returns 1 (since an image is a table with 1 column and 1 row).

Context 3 (R1.8): Return an error if the table is being filtered.

Context 4 (R3): Notwithstanding the comments in Context 1, support double-filtering to get the right answer in the case of a filtered table.

4.13.9 dmTableGetRow

long dmTableGetRow(dmBlock* table, void* row)

(R1): Fill the supplied row buffer (either a structure or an array) with data from the current row of the table, returning the row number of the retrieved row and advancing the row pointer by one. The value dmNOMOREROWS is returned when the end of table is reached, or on error.

4.13.10 dmTableGetRowNo

long dmTableGetRowNo(dmBlock* table)

Context 1 (R1): Table: Returns the current row number, or dmBADROW on error.

Context 2 (R1): Image: Returns 1.

4.13.11 dmTableNextRow

long dmTableNextRow(dmBlock* table)

(R1): Advance the row pointer so that future reads will come from the next row of the table. dmNOMOREROWS will be returned on error or when the end of the table is reached, otherwise the new current row number is returned. This routine causes an error for an image.

4.13.12 dmTableOpen: Open table and dataset

dmBlock* dmTableOpen(char* vdataSpec)

Opens a table in a dataset. Used when you're only interested in one table in the dataset - avoids having to make separate calls to handle the dataset and table.

Context 1 (R1): Open a table datablock, and its dataset. The dataset is opened and then the named table is opened, the supported syntax being "dataset[tablename]". Note that a dmBlock* is returned, NOT a dmDataset*, as often the user will not want to care about the dataset. If a dmDataset* is needed, use the individual dataset and table open calls.

Context 2 (R1.7): If no block name is specified, then the first interesting block in the dataset will be opened. The definition of 'first interesting' block is kernel-dependent; for FITS, it is the first block with NAXIS not equal to zero that is also not a GTI.

R1.7: Renamed, was dmDatasetTableOpen.

Context 3 (R1.5): Open a virtual table datablock, and its dataset. The virtual data specification is parsed according to the filtering syntax in the ASC Filter Syntax document. The underlying physical dataset is opened, after which filters are set up to provide the virtual view of the ASC dataset. The kernel is determined at open time from the underlying file contents (not the file name, although that may be used as a hint). Note that if you want to open another Virtual Table using the same underlying file, the two ASC Datasets are completely independent, but at the kernel layer you may want to only open the underlying file once.

4.13.13 dmTableOpenColumn: Get column handle

dmDescriptor* dmTableOpenColumn(dmBlock* table, char *colName)

Context 1 (R1): Searches in table for a column with the given name, returning a column descriptor (NULL if not found). In (R1.5) the comparison will be case-insensitive, and the first match found (in order of column number) is returned. No check for ambiguity is performed.

Context 2 (R1.7): Return a descriptor for a vector column's component if the component name matches and there is no match for any of the full descriptor names.

Context 3 (R2.1): For an image, this routine is pretty useless. It should return the image data descriptor if you give it the image data name, but that's not a high priority for implementation.

Context 4 (R3): We should support wild cards in the name search, like FITSIO does.

4.13.14 dmTableOpenColumnList: Get list of columns

dmDescriptor** dmTableOpenColumnList(dmBlock* table, long* ncols)

Context 1 (R1): Returns a list of data descriptors, one for each column in the table. The number of descriptors returned is indicated by the numcols parameter. This wraps the functionality of dmTableGetNoCols and dmTableOpenColumnNo in a convenient way. The user must free the array memory (but not the array contents).

Context 2 (R1): If the table was opened with dmTableOpenSelect, only the selected columns are returned.

Context 3 (R1): If the block is actually an image, ncols = 1 and the single descriptor is that of the image data.

4.13.15 dmTableOpenColumnNo: Get column handle

dmDescriptor* dmTableOpenColumnNo(dmBlock* table, long colNo)

Context 1 (R1): Return the data descriptor for the nth column in the table. Returns null if column number is out of range.

Context 2 (R1): For an image, returns the image data descriptor if colNo = 1, otherwise returns null.

Context 3 (R1.7): Note that unlike dmTableOpenColumn, this routine doesn't see individual components of a vector column.

4.13.16 dmTableOpenSelect: Select row structure

dmBlock* dmTableOpenSelect(dmDataset* dataset, char* tabname, char* columnlist, dmDataType *castToType)

Context 1 (R1): Select a subset of the given tables columns for opening, as specified by the comma-delimited columnlist. Note that if you are going to perform row-based I/O on the table (via dmTableGetRow), your row structure must contain ONLY fields that correspond to the selected columns, with care being taken that the structure and column datatypes match. One example where this routine proves useful is that it gives one the ability to read GTI tables from both Einstein and ROSAT archives, using the same piece of code. Since both tables will contain "START" and "STOP" columns, the reader code can select just those two for opening, effectively ignoring any other columns that might be present (thus improving code usability across the archives).

Context 2 (R1.5): Passing in a pointer to a variable of type dmDataType allows the caller to specify that on subsequent entire-row based reads (again, via dmTableGetRow) all column values will be cast to the specified type. This is useful when you do not know prior to compilation how many columns you'll eventually need to scan. In this instance, instead of passing a pointer to a C structure to dmTableGetRow, the caller passes an array of type sufficiently large to store the range of possible column values, and size sufficiently large to contain the largest expected number of columns. Note that ONLY strictly numeric types will be accepted as possible cast types, or NULL to indicate no casting is desired.

Context 3 (R1.9): As above, but support a modified syntax to the columnlist which allows forcing of the data types (in case the data types are different from what is expected). A possible modified syntax is (in the spirit of PROS eventdef) to optionally follow column names by a colon and a letter indicating the type they will have in the row struct - for instance, "START:d,STOP:d" to indicate two doubles. This proposed functionality is subject to further design review.

This routine will become less needed when filtering is fully implemented, as one may then just use dmBlockOpen with the appropriate column selection enforced. The routine throws an error if used with an image block.

4.13.17 dmTablePutRow

long dmTablePutRow(dmBlock* table, void* row)

(R1): Close out this row of table, advancing to the next row for future write operations. If the row structure pointer is null, assume all writes have already been done using dmSetScalar etc. Otherwise, write values using the current row-based I/O structure.

4.13.18 dmTableSetRow

long dmTableSetRow(dmBlock* table, long rowNo)

(R1) Go to the specified absolute row number in the given table and return the current row number value. If rowNo $<$ 1, the row pointer will be set to 1. If rowNo $>$ number of rows in the table, or if dmENDOFTABLE is specified, the row pointer will be adjusted to point past the last table row, and dmNOMOREROWS will be returned. In this way, new rows may be easily added to the end of an existing table. For all other error conditions, dmBADROW will be returned. Note that images have only 1 row.

5 Changes to Abstract Data Model Design

In reviewing the API, it became clear that some changes to the abstract design are needed. These changes will later be folded in to the abstract design document.

5.1 Element Type Component Names

Following discussions on the common data model mail exploder and review of actual examples, it turns out that we need to add component names to the component columns of the Element Types. In the earlier design, we assumed that a dmINTERVAL element called X would get automatically assigned kernel column names X, X_MIN, X_MAX. It turns out that it's better to allow the user to specify these names. We now converge the idea of vector columns and element types somewhat, considering them as one one-dimensional list of components: e.g. a column of element dimension 2 and element type dmInterval has six components X, X_MIN, X_MAX, Y, Y_MIN, Y_MAX. However, we retain the distinction between the vector components and the element type components: the element type components are assigned a special meaning (e.g. the minimum of a range), which allows the filtering process to work.

5.2 Comments

The support for FITS type COMMENT header info has been changed; see dmBlockWriteComment and dmBlockReadComment.

6 Future Requirements

6.1 Further formats

The IRAF-QPOE kernel should also support PLMASK files and ST TABLE files. These should be read automatically and written using a kernel hint option. There should be an ASCII kernel which supports simple ASCII files and John Roll's Starbase format.