CXC-DM-009

# CXC Data Model

# Vol. 9

# C Programmers' Reference

Chandra X-ray Center

October 22, 2001

# Contents

# 1 Introduction

This document lists the functions provided by the CXC 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. (R1.9) for release 1.9.

It is assumed that readers are familiar with the C Progammers' Guide which describes use of the library in more general terms.

# 2 dmArray Routines

## 2.1 dmArrayCreateAxisGroup

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

Currently only type=dmDOUBLE will work.

(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.

### 2.2 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. Axis groups are counted starting at 1.

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.

### 2.3 dmArrayGetNoAxisGroups

long dmArrayGetNoAxisGroups( dmDescriptor* imageData )

Context 1 (R1.7): Return number of axis groups.

Context 2 (R3.0): Support for table cell images.

## 3 dmBlock Routines

### 3.1 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.

### 3.2 dmBlockClose

int dmBlockClose(dmBlock* block)

(R1): Close a datablock within the dataset, but keep the dataset open. Returns dmSUCCESS on success; otherwise use dmGetErrorMessage.

## 3.3 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.

## 3.4 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.

## 3.5 dmBlockCreateSubspaceCpt

int dmBlockCreateSubspaceCpt(dmBlock* block)

(R1.6): Create a new component of the data subspace of the block (i.e. make a new row in the virtual data subspace table). Set the current subspace component to be this new component. Returns dmSUCCESS unless there is an internal problem.

### 3.6 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. The routine returns dmSUCCESS except on failure.

### 3.7 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.

### 3.8 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.

### 3.9 dmBlockGetKernelKey

int dmBlockGetKernelKey(dmBlock* table, long n, char* buf, long maxlennn)

(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, for some definition of header.) Diagnostic routine to bypass data model interpretation. You are not guaranteed that the string returned will bear any particular relation to the format in which it is stored in the real file.

### 3.10 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.

### 3.11 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.

### 3.12 dmBlockGetName

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

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

### 3.13 dmBlockGetNo: Get block no

long 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. On error, the routine returns zero.

## 3.14 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.

## 3.15 dmBlockGetNoKeys

long dmBlockGetNoKeys(dmBlock* block)

(R1): Gets the number of header keys in the block. Returns -1 on error.

## 3.16 dmBlockGetNoSubspaceCols

long dmBlockGetNoSubspaceCols(dmBlock* block)

(R1.6) Get the number of subspace columns in the data subspace. On error, the routine returns -1.

## 3.17 dmBlockGetNoSubspaceCpts

long dmBlockGetNoSubspaceCpts(dmBlock* block)

(R1.6) Get the number of components in the block's data subspace. Always returns 0 or more.

## 3.18 dmBlockGetPrefAxes: Get preferred column or axis

dmDescriptor** dmBlockGetPref(dmBlock* block, dmDescriptor** weight, long* ncols)

(R1.8): The routine returns the number of preferred axes (in ncols) and a list of their descriptors as the return value. It also returns a descriptor which is the 'weight' column, if any (if there is no weighting, *weight = NULL). The list of descriptors must be freed by the user (but the entries in the list should not be). Example:

```
dmDescriptor** list;
dmDescriptor*  weight;
long           npref;
```

```
list = dmBlockGetPref( block, &weight, &npref );
...
free( list );
```

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.

(R1.8): The ‘weight’ argument 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.

## 3.19 dmBlockGetSubspace

dmDescriptor** dmBlockGetSubspace(dmBlock* block,long* ncols,long* 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).

## 3.20 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.

### 3.21 dmBlockGetType

dmBlockType dmBlockGetType(dmBlock* block)

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

### 3.22 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.

### 3.23 dmBlockInSubspace: Test whether in data subspace

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

(R2.1): Returns true if the given 'row' of n items has values which lie in the data subspace of block.

### 3.24 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.

### 3.25 dmBlockMergeSubspace

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

(R1.8): Take the subspaces of blocks 1 and 2 and perform the logical OR, putting the result in block0. Blocks need not be unique, e.g. block 0 and block2 may be the same. Returns dmSUCCESS or dmFAILURE.

### 3.26 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.

### 3.27 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. Note that 0 is a valid value for the argument, positioning the header prior to the first DataModel card.

### 3.28 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 (R1.8): Filtered block open. As above, but support a virtual block specification (filtered block using DataModel 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.

## 3.29 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.

## 3.30 dmBlockPrintSubspace: List data subspace

int dmBlockPrintSubspace( dmBlock* block )

(R1.8): Diagnostic routine. List the data subspace associated with the given block, printing a description of it to standard output.

## 3.31 dmBlockReadComment

dmDescriptor* 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.

This routine is currently buggy; I hope to fix this in release 2.0.

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

```
dmBlock* block;
long i,n;
char* tag;
char* comment;
dmDescriptor* key;
char name[MAXLEN];
block = dmTableOpen( "bas.fits[stdevt]" );
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 );
  }
}
dmTableClose( block );
```

## 3.32 dmBlockSetPref

int dmBlockSetPref(dmBlock* block, char* prefspec )

(R1.8): Set the preferred axes for an image or a table. The input arguments include a string which will be written to the CPREF header key. The preferred axes are a comma-separated list of column names,.e.g "DETX,DETY". An alternate format also specifies a weighting function, e.g. "PHA(DETX,DETY)" which indicates that a PHA weighted image is the default binning result. However, weighted binning has not been implemented yet.

(R2.0) In the QPOE kernel, sets the QPOE keys to be the first two preferred axes.

(R3) Implement weighted binning.

## 3.33 dmBlockSetSubspaceCpt

int dmBlockSetSubspaceCpt( dmBlock* block, long 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.

## 3.34 dmBlockWriteComment

dmDescriptor* 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.

The value returned by this function is the key with which the comment is associated. The return value type may be changed in later releases, but is guaranteed to be 0 or NULL for failure.

# 4 dmColumn Routines

## 4.1 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.

## 4.2 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 dmColumnCreateElement

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

Context 1 (R2.1): 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).

## 4.4 dmColumnCreateNDArray

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

(R1.8): 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.

## 4.5 dmColumnCreateVarArray

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

(R3): Special routine to create a 1D array column, which will be stored as a variable-length array if the underlying 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.

Unfortunately, the current design of the DM doesn't lend itself to supporting this routine, so its implementation has been delayed.

## 4.6 dmColumnCreateVector

dmDescriptor* dmColumnCreateVector( dmBlock* table, char* name, dmDataType type, long strlen, 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.

## 4.7 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 )

(R1.9): 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 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 dmColumn-CreateElement. In R1.9, only dmVALUE elements (the simplest kind) are supported.

(R2.1): Support for other element types.

## 4.8 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.

## 4.9 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.

## 4.10 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.

# 5 dmCoord Routines

## 5.1 dmCoordCalc

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

Context 1 (R1.8): Using the specified coordinate descriptor, return the transformed value corresponding to the given input. The input type is up to the user, but transformed output value is always double. A set of routines is provided for both scalar and vector descriptors, so the input and return arguments are pointers. The parent of the input coordinate descriptor is a table column.

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

Context 3 (R2.1): The parent of the input descriptor 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 (or image). To read the columns of a table returning the transformed coordinate values, make the usual calls to the column read routines, but pass the coordinate descriptor instead of the parent column data descriptor.

Example:

```
dmDescriptor* pos = dmTableOpenColumn(table,"POS");     /* Vector column, long datatype */
dmDescriptor* world = dmDescriptorGetCoord( 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 = dmDescriptorGetCoord( pha );
double value2 = 4.8;                                    /* Even though the column is integral */
double result2;
dmCoordCalc_d( energy, &value2, &result2 );
```

## 5.2 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)

(R3) 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.

## 5.3 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, long* 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.

To automatically transform the column values on read, pass to the routines (dmGetScalar_*, etc) the coordinate descriptor created here, rather than the parent column descriptor.

The arguments are:

- dd: the parent descriptor.
- name: name of the coordinate descriptor

- unit: unit of the coordinate descriptor.
- dim: the element dimension of the coordinate descriptor which must be the same as the parent descriptor, or an error is set.
- cptnames: names of coordinate components for 2-dim coordinates (e.g. 'RA', 'DEC'). Null for 1-D case.
- 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: (R2.1) As above, but parent descriptor is an image data descriptor.

Context 4: (R2.1) 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.

## 5.4 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 )

(R3) 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.

## 5.5 dmCoordCreateLinear

int dmCoordCreateLinear_s( dmDescriptor* dd, char* name, char* unit,short crpix, double crval, double cdelt )

int dmCoordCreateLinear_l( dmDescriptor* dd, char* name, char* unit,long crpix, double crval, double cdelt)

int dmCoordCreateLinear_f( dmDescriptor* dd, char* name, char* unit,float crpix, double crval, double cdelt)

int dmCoordCreateLinear_d( dmDescriptor* dd, char* name, char* unit,double crpix, double crval, double cdelt)

int dmCoordCreateLinear_us( dmDescriptor* dd, char* name, char* unit,unsigned short crpix, double crval, double cdelt)

int dmCoordCreateLinear_ul( dmDescriptor* dd, char* name, char* unit,unsigned long crpix, double crval, double cdelt)

int dmCoordCreateLinear_ub( dmDescriptor* dd, char* name, char* unit,unsigned char crpix, double crval, double cdelt)

(R1.7) This makes the common case of creating a 1D linear coordinate transform easier, as you can use literal constants instead of having to create a variable and pass a pointer to it to the more general dmCoordCreate_* routines. For example:

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

## 5.6 dmCoordCreateLookup

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

(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).

## 5.7 dmCoordGetAxisGroupNo

long dmCoordGetAxisGroupNo(dmDescriptor* dd)

(R1.7): Return the axis group number (within an image) of the given axis group coordinate descriptor.

## 5.8 dmCoordGetParams

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

(R2.1) Returns 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 will almost always be zero.

## 5.9 dmCoordGetParent

dmDescriptor* dmCoordGetParent(dmDescriptor* dd)

(R1.7) Get the parent descriptor to which the given coordinate is attached, returning null if the descriptor is not a coordinate descriptor (the inverse of dmDescriptorGetCoord routine).

## 5.10 dmCoordGetTransform

int dmCoordGetTransform_s(dmDescriptor* dd,short* crpix,double* crval,double* cdelt,long dim)
int dmCoordGetTransform_l(dmDescriptor* dd,long* crpix,double* crval,double* cdelt,long dim)
int dmCoordGetTransform_f(dmDescriptor* dd,float* crpix,double* crval,double* cdelt,long dim)
int dmCoordGetTransform_d(dmDescriptor* dd,double* crpix,double* crval,double* cdelt,long dim)
int dmCoordGetTransform_ub(dmDescriptor* dd,unsigned char* crpix,double* crval,double* cdelt,long dim)
int dmCoordGetTransform_us(dmDescriptor* dd,unsigned short* crpix,double* crval,double* cdelt,long dim)
int dmCoordGetTransform_ul(dmDescriptor* dd,unsigned 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 dim is passed to the routine for safety, to ensure array bounds are not exceeded (at most dim values will be returned).

### 5.11 dmCoordGetTransformType

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

(R1.7): Return the transform type for a coordinate transform. The input argument must be a true cordinate descriptor, not an axis group coordinate descriptor. The number of transform parameters is returned; this is usually zero.

### 5.12 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. An error code is returned if the inverse mapping is single-valued, not meaningful, or if the value is outside the range of the mapping.

### 5.13 dmCoordSetTransform

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

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

# 6 dmDataset Routines

## 6.1 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.

## 6.2 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 DataModel 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.

## 6.3 dmDatasetClose (Close dataset)

int dmDatasetClose(dmDataset* ds)

(R1) Close a DataModel 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.

## 6.4 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" );
```

### 6.5 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.

### 6.6 dmDatasetCreateImage

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

(R1.7) Create an empty image in the dataset, with the specified dimensions.

### 6.7 dmDatasetCreateTable

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

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

### 6.8 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.

### 6.9 dmDatasetDestroy

int dmDatasetDestroy(char* dsname )

This routine deletes an unopened dataset. It is used to clobber an old output file prior to creating a new one.

## 6.10 dmDatasetGetBlockName

int dmDatasetGetBlockName(dmDataset* ds,long blockNo,char* blName,long 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.

## 6.11 dmDatasetGetBlockType

dmBlockType dmDatasetGetBlockType(dmDataset* ds, long 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.

## 6.12 dmDatasetGetCurrentBlockNo (Get current block number)

long 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.

## 6.13 dmDatasetGetKernel

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

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

## 6.14 dmDatasetGetName

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

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

## 6.15 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.

## 6.16 dmDatasetGetNoBlocks (Get number of blocks)

long 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.

## 6.17 dmDatasetMoveToBlock (Direct access to block)

dmBlock* dmDatasetMoveToBlock(dmDataset* ds, long 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.

Do not attempt to reopen a filtered block without first reopening the dataset itself, or unpredictable results will be obtained.

## 6.18 dmDatasetOpen (Open dataset)

dmDataset* dmDatasetOpen(char* datasetName)

Context 1 (R1): Open an existing dataset read-only 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 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.

## 6.19 dmDatasetOpenUpdate (Open dataset for update)

dmDataset* dmDatasetOpenUpdate(char* datasetName)

As dmDatasetOpen, but attempts to open the dataset read/write.

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

## 6.20 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.

## 6.21 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.

### 6.22 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.

## 7 dmDescriptor Routines

### 7.1 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.

### 7.2 dmDescriptorDelete

int dmDescriptorDelete(dmDescriptor* dd)

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

## 7.3 dmDescriptorGetBin

```
int dmDescriptorGetBin_s(dmDescriptor* dd, short* value)
int dmDescriptorGetBin_l(dmDescriptor* dd, long* value)
int dmDescriptorGetBin_f(dmDescriptor* dd, float* value)
int dmDescriptorGetBin_d(dmDescriptor* dd, double* value)
int dmDescriptorGetBin_ub(dmDescriptor* dd, unsigned char* value)
int dmDescriptorGetBin_ul(dmDescriptor* dd, unsigned short* value)
int dmDescriptorGetBin_ul(dmDescriptor* dd, unsigned long* value)
```

Context 1 (R1.8): Get the default binning factor for the column. Its type is the same as the type of the column. If no binning has been specified, sets *value to 1 and returns dmFAILURE.

## 7.4 dmDescriptorGetBlock

```
dmBlock* dmDescriptorGetBlock(dmDescriptor* dd)
```

Context 1 (R1.7): Return the parent block to which the descriptor belongs.

## 7.5 dmDescriptorGetCoord

```
dmDescriptor* dmDescriptorGetCoord(dmDescriptor* dd)
```

Context 1 (R1.7): Return the first coordinate descriptor associated with the given descriptor, or NULL if none such exists.

## 7.6 dmDescriptorGetCoordNo

```
dmDescriptor* dmDescriptorGetCoordNo(dmDescriptor* dd,long coordNo)
```

(R1.7): Return the nth coordinate transform descriptor associated with the given descriptor, or NULL if none such exists.

## 7.7 dmDescriptorGetLength

long dmDescriptorGetLength(dmDescriptor* dd)

This routine gets the value of the length parameter for those data types for which it is defined, namely string and bit types. It was formerly known as dmGetStrLen.

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 (R1.74): Get the number of bits in a dmBIT table column. For any other data type, return zero.

## 7.8 dmDescriptorGetNoCoords

long dmDescriptorGetNoCoords(dmDescriptor* dd)

(R1.7): Get the number of coordinate transform descriptors associated with the given descriptor.

## 7.9 dmDescriptorGetNull

int dmDescriptorGetNull_s(dmDescriptor* dd, short* value)
int dmDescriptorGetNull_l(dmDescriptor* dd, long* value)
int dmDescriptorGetNull_f(dmDescriptor* dd, float* value)
int dmDescriptorGetNull_d(dmDescriptor* dd, double* value)
int dmDescriptorGetNull_ub(dmDescriptor* dd, unsigned char* value)
int dmDescriptorGetNull_ul(dmDescriptor* dd, unsigned short* value)
int dmDescriptorGetNull_ul(dmDescriptor* dd, unsigned long* value)
int dmDescriptorGetNull_c(dmDescriptor* dd, char** value)

Context 1 (R1.8): Get the null value for the column. Its type is the same as the type of the column. If no null has been specified, sets *value to 0 and returns dmFAILURE. For float and double cases, this default implies that NaN should be considered the null value.

## 7.10 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)
int dmDescriptorGetRange_ub(dmDescriptor* dd, unsigned char* min, unsigned char* max)
int dmDescriptorGetRange_us(dmDescriptor* dd, unsigned short* min, unsigned short* max)

int dmDescriptorGetRange_ul(dmDescriptor* dd, unsigned long* min, unsigned long* max)
int dmDescriptorGetRange_bit(dmDescriptor* dd, unsigned char* min, unsigned char* 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 columns of type double we also recognize the possibility of special syntax (i.e. TSTART and TSTOP keys).

If no range keywords have been defined for the column, then the minimum column value will be set to -DATA_TYPE_MAX, and the maximum will be set to DATA_TYPE_MAX, where DATA_TYPE_MAX is the maximum value allowed by datatype of the column (eg, LONG_MAX or DBL_MAX).

(R1.9) Support for subspace descriptors if they have associated columns, by propagating the result from the column.

## 7.11 dmDescriptorGetType

dmDescriptorType dmDescriptorGetType(dmDescriptor* dd)

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

## 7.12 dmDescriptorInfo: Get all descriptor info

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

(R3) An alternate way of getting the descriptor info, retrieving all descriptor attributes at once. *Note changes to calling sequence 1997 Jul 20.*

## 7.13 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.

## 7.14 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.

## 7.15 dmDescriptorPrintList: List descriptors

int dmDescriptorPrintList( dmBlock* table )

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

## 7.16 dmDescriptorSetBin

int dmDescriptorSetBin_s(dmDescriptor* dd, short value)
int dmDescriptorSetBin_l(dmDescriptor* dd, long value)
int dmDescriptorSetBin_f(dmDescriptor* dd, float value)
int dmDescriptorSetBin_d(dmDescriptor* dd, double value)
int dmDescriptorSetBin_ub(dmDescriptor* dd, unsigned char value)
int dmDescriptorSetBin_ul(dmDescriptor* dd, unsigned short value)
int dmDescriptorSetBin_ul(dmDescriptor* dd, unsigned long value)

Context 1 (R1.8): Set the default binning factor for the column. Its type must be the same as the type of the column. In FITS, this writes the TDBINn keyword.

## 7.17 dmDescriptorSetNull

int dmDescriptorSetNull_s(dmDescriptor* dd, short value)
int dmDescriptorSetNull_l(dmDescriptor* dd, long value)
int dmDescriptorSetNull_f(dmDescriptor* dd, float value)
int dmDescriptorSetNull_d(dmDescriptor* dd, double value)
int dmDescriptorSetNull_ub(dmDescriptor* dd, unsigned char value)
int dmDescriptorSetNull_ul(dmDescriptor* dd, unsigned short value)
int dmDescriptorSetNull_ul(dmDescriptor* dd, unsigned long value)
int dmDescriptorSetNull_c(dmDescriptor* dd, char* value)

Context 1 (R1.8): Set the null value for the column. Its type must be the same as the type of the column, In FITS, this writes the TNULLn keyword, except for the float and double cases (where this is illegal) when it writes a (local DM convention) TDNULLn.

For float and double columns, using NaN as the null value is the default in FITS, rather that writing an explicit value to the header, and use of the SetNull_f and SetNull_d routines should be avoided in CXC code - they are provided on an experimental basis.

### 7.18 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)
int dmDescriptorSetRange_us(dmDescriptor* dd, unsigned short min, unsigned short max)
int dmDescriptorSetRange_ul(dmDescriptor* dd, unsigned long min, unsigned long max)
int dmDescriptorSetRange_ub(dmDescriptor* dd, unsigned char min, unsigned char max)
int dmDescriptorSetRange_bit(dmDescriptor* dd, unsigned char min, unsigned char 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.

## 8 dmFile Routines

A special routine to provide POSIX-compliant file access information.

### 8.1 dmFileExists

int dmFileExists( char* filename )

This routine tests access for a file. The return values are:

| | |
|---|---|
| DMF_READWRITE | Read and write permission |
| DMF_READONLY | Read only permission |
| DMF_NOPERM | No read permission |
| DMF_NOFILE =0 | File does not exist |

The value of DMF_NOFILE is guaranteed to be 0, so that `if( dmFileExists( filename ) )` may be used as a test of existence.

# 9 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.

## 9.1 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_bit(dmDescriptor* dd, unsigned char* vals,long npixels
int dmGetArray_us(dmDescriptor* dd,unsigned short* values,long npixels)
int dmGetArray_ul(dmDescriptor* dd,unsigned long* values,long npixels)

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.

## 9.2 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.

## 9.3 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.

## 9.4 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.

## 9.5 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. This may then be used in calls to dmGetScalar, etc, to access data of the specified component column. If cptNo is more than the number of components, return NULL.

As currently scheduled, R1.9 will support direct I/O on the vectored column descriptor with the dmGetVector* routines.

Context 2 (R1.8): If the element dimension of the argument is 1 (ie, the dd is a scalar descriptor), return the descriptor itself.

Context 3 (R1.8): If the argument is a coordinate descriptor of dimension more than 1, return NULL, since getting only one component of a 2D transform is a bad idea.

## 9.6 dmGetCptName

int dmGetCptName(dmDescriptor* dd,long cptNo,char* cptName,long 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.

## 9.7 dmGetDataType

dmDataType dmGetDataType(dmDescriptor* dd)

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

## 9.8 dmGetDataTypeName

char* dmGetDataTypeName(dmDescriptor* dd)

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

## 9.9 dmGetDesc

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

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

## 9.10 dmGetDisp: Get descriptor display format

int dmGetDisp(dmDescriptor* dd, char* display, long 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.

## 9.11 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: The specification for this routine changed between R1.6 and R1.7.

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.

## 9.12 dmGetElementDim

long dmGetElementDim(dmDescriptor* dd)

Returns the number of vector components for the descriptor. In the DataModel 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.

## 9.13 dmGetElementType

dmElementType dmGetElementType(dmDescriptor* dd)
int dmGetElementTypeName(dmDescriptor* dd, char* name, long maxlen )

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.

## 9.14 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.

## 9.15 dmGetErrorMessage

void dmGetErrorMessage(char* errMessage, long maxlen)

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

## 9.16 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.

## 9.17 dmGetName

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

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

## 9.18 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, long maxlen)
dmBool dmGetScalar_q(dmDescriptor* dd)
unsigned char dmGetScalar_ub(dmDescriptor* dd)
int dmGetScalar_bit(dmDescriptor* dd, void* value)
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.

## 9.19 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, long maxlen,long firstRow,long nrows)
long dmGetScalars_ub(dmDescriptor* dd, unsigned char* vals, long firstRow, long nrows)
long dmGetScalars_bit(dmDescriptor* dd, unsigned char* vals,long firstRow, long nrows)
capilong dmGetScalars_us(dmDescriptor* dd, unsigned short* vals, long firstRow, long nrows

long dmGetScalars_ul(dmDescriptor* dd, unsigned long* vals, long firstRow, long 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 **dmDescriptorGetLength** 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.

## 9.20 dmGetUnit

int dmGetUnit(dmDescriptor* dd, char* unit, long 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.

If the descriptor has no attached unit, the unit string will be blanked out upon return.

## 9.21 dmGetVector: Get vector value

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

Context 1 (R1.8): 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 (R1.8): Cast type if necessary (dd data type and routine type in disagreement).

Context 3 (R1.8): Work correctly for scalar case (returns one value, effect is same as dmGetScalar).

Context 4 (R1.8): Return value for vectored coordinate descriptor, by calling dmGetVector for the parent descriptor and then applying the coord transform. (Implemented for dmGetVector_d only).

Context 5 (R2): Work for vectored header keys.

### 9.22 dmGetVectors: read cells from multiple table rows

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

### 9.23 dmGetVersion

int dmGetVersion(char* versionString, long maxlen)

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

## 10 dmImage Routines

### 10.1 dmImageClose

int dmImageClose(dmBlock* image)

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

## 10.2 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.

## 10.3 dmImageDataGetPixel

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

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

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

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.

## 10.4 dmImageDataGetPixlist: Get pixel list

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

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.

## 10.5 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.

## 10.6 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).

In a 2D image, the offset into the array for pixel (i,j) is n * (j-1) + i-1, where n is the size of axis 1 (i.e. axes[0] from dmGetArrayDimensions).

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.

## 10.7 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) ).

## 10.8 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. For 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.

## 10.9 dmImageDataSetPixlist

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

DataModel 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.

## 10.10 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.

## 10.11 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.

## 10.12 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): Open a dataset and filtered image. Currently only image subsection filters (ie, explicit axis pixel lists) are supported. Region filtering will be supported in a later release.

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.

### 10.13 dmImageOpenUpdate

dmBlock* dmImageOpenUpdate(char* vdataSpec)

This routine is like dmImageOpen, but if no filter is specified it opens the block read/write. We don't support read/write access through a filter.

## 11 dmInit

### 11.1 dmInit

void dmInit(int argc,void *argv)

dmInit performs necessary DM layer and kernel initialization, and accepts the traditional command-line argument argc and argv as input parameters (but doesn't currently do anything with them). It is not required that end-user programs call dmInit explicitly, as the DM library implicitly ensures that it is called once in all DM programs (multiple dmInit calls have no effect).

If your application contains custom initialization code, it is generally a good idea to include a call to dmInit as early in that code as possible, so that application-specific customizations (like signal handlers) are not inadvertently overridden by the DM.

## 12 dmKernel Routines

### 12.1 dmKernelGetCopy: Get copy kernel

int dmKernelGetCopy(char* copyKernelName, long maxlen)

(R1): Return the current copy kernel name.

### 12.2 dmKernelGetCreate: Get creation kernel

int dmKernelGetCreate(char* creationKernelName, long maxlen)

(R1): Return the current creation kernel name.

### 12.3 dmKernelGetList

int dmKernelGetList(char** kernelIDs, long* 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.

### 12.4 dmKernelSetCopy: Set copy kernel

char* dmKernelSetCopy(char* kernelId)

(R1): Specify the kernel to associate with an opened virtual block being copied (eg, by dmBlockCreateCopy). The default behaviour is for the copy to retain the kernel of the underlying input dataset; this routine overrides the default. Currently accepted kernelID values are given in section **??**

### 12.5 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.

### 12.6 dmKernelSetOption

int dmKernelSetOption(char* option)

(R1.96): Set an internal kernel parameter. This mechanism will be used to tune particular kernel parameters. At the moment TABLE options are defined, and are given in **??**. If called with a kernel option that has no meaning for the current kernel, the option is simply ignored.

## 13 dmKey Routines

### 13.1 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.

## 13.2 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.

## 13.3 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.

## 13.4 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.

## 13.5 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,long 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.

## 13.6 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 maxlen, long dim, long* nvals )
dmDescriptor* dmKeyReadVector_q( dmBlock* block, char* name, dmBool* 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.

## 13.7 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,dmBool 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).

## 13.8 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.
```

## 13.9 dmKeyWriteElement: Write Element header key

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

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.* Nvals indicates the number of elements in each value set.

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

## 13.10 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, dmBool* 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.

# 14 dmSet Routines

## 14.1 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)
int dmSetArray_bit( dmDescriptor* dd,unsigned char* 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 (R1.8): Write data to an image block (set value of image data descriptor).

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

## 14.2 dmSetCptName

void 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.

## 14.3 dmSetDesc

void 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.

## 14.4 dmSetDisp

void dmSetDisp(dmDescriptor* dd, char* dispFormat)

(R1.8): 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.

## 14.5 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.

### 14.6 dmSetError

void dmSetError( int code, char* ErrMessage )

Set the DM error status; this is really an internal routine.

### 14.7 dmSetInternals: Modify internal DM behavior

int dmSetInternals(char* paramname, void* paramvalue)

Modify the behavior of various DM internals. For example, using this routine one can fix the size of the internal table buffer, ala

```
long bufsize = 5000;
dmSetInternals(dmTABLEBUFFERSIZE,&bufsize);
```

The complete list of internal parameters that may be modified is given in ??.

### 14.8 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.

### 14.9 dmSetName

int dmSetName(dmDescriptor* dd, char* name)

(R1.9) Set or change the name of the descriptor, irrespective of descriptor type.

## 14.10 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, dmBool 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 )
int dmSetScalar_bit( dmDescriptor* dd, unsigned char 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).

## 14.11 dmSetScalars: write cells to multiple table rows

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

long dmSetScalars_br(dmDescriptor* dd, char** value, int firstRow, long nrows)

long dmSetScalars_ub(dmDescriptor* dd, unsigned char* value, int firstRow, long nrows)

long dmSetScalars_us(dmDescriptor* dd, unsigned short* value, int firstRow, long nrows)

long dmSetScalars_ul(dmDescriptor* dd, unsigned long* value, int firstRow, long nrows)
long dmSetScalars_bit(dmDescriptor* dd, unsigned char* value, int firstRow, long 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.

## 14.12 dmSetUnit

int dmSetUnit(dmDescriptor* dd, char* unit)

Set the (case sensitive) 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’.

(R1.8) Works on header keys.

## 14.13 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, dmBool* 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.

### 14.14 dmSetVectors: write cells to multiple table rows

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

### 14.15 dmSetVerbose

void dmSetVerbose( int verbose )

Controls printing of internal status and error messages to stderr. Default is verbose = 0 (no output). Verbose can be 0 to 5.

## 15 dmSubspace Routines

### 15.1 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.

## 15.2 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* name, char** cptNames, 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).

### 15.3 dmSubspaceColGet

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

Get the values of a subspace column in the current subspace component. This routine returns allocated arrays of values and the number of such values. User must free the memory.

The first argument must be a subspace descriptor. Note that to get ranges for a 2-dimensional subspace descriptor (e.g. for a spatial coordinate pair) you can use dmGetCpt to get the subspace descriptors for the components, and then call dmSubspaceColGet on the resulting component descriptors.

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.

### 15.4 dmSubspaceColGetTableName

int dmSubspaceColGetTableName(dmDescriptor* col, char* name, long maxlen)

(R1.72): Get the name of the value table for the given data subspace descriptor. Sets name to the null string if there is no value table.

### 15.5 dmSubspaceColIntersect

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

(R2.1) 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).

### 15.6 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 dmTableOpen-Column and dmKeyOpen. (*new routine*).

## 15.7 dmSubspaceColRead

dmDescriptor* dmSubspaceColRead_s( dmBlock* block, char* name, short** value1, short** value2, long* nvalues )
dmDescriptor* dmSubspaceColRead_l( dmBlock* block, char* name, long** value1, long** 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.

## 15.8 dmSubspaceColSet

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

## 15.9 dmSubspaceColSetTableName

int dmSubspaceColSetTableName(dmDescriptor* col, char* name)

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

## 15.10 dmSubspaceColUpdate

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

### 15.11 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.

### 15.12 dmSubspaceGetRegion

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

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

### 15.13 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.

## 16 dmTable Routines

### 16.1 dmTableAllocRow

void* dmTableAllocRow(dmBlock* table)

(R1.6): This routine supports row-based I/O on tables whose structure you don't know prior to runtime. In particular, it lets you copy rows from one table to another without needing to know anything about the columns the row contains. The routine should be called after you open the table but before you start processing the rows; it returns a void pointer to a row structure (allocated internally with the appropriate size and structure) which can contain the data. This row structure may be passed to the dmGetRow and dmPutRow calls.

(R1.7) Supplemental routines have been added to the API (dmTableGetColOffset and dmTableGetColPtr, see below) which let you access the individual columns within the void* row structure returned here.

## 16.2 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.

NOTE: In R1.7 this function was renamed, from dmDatasetTableClose.

## 16.3 dmTableCopyRow

int dmTableCopyRow( dmBlock* table1, dmBlock* table2 )

(R2.1): 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'.

## 16.4 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]," which is compatible with the general DM filtering syntax. The dataset pointer is not directly returned, but may be accessed by the dmBlockGetDataset routine. After creation, the table has zero rows columns. Use dmDatasetCreateTable to create a table in an existing dataset.

NOTE: In R1.7 this function was renamed, from dmDatasetTableCreate.

## 16.5 dmTableCreateColumns

dmDescriptor** dmTableCreateColumns( dmBlock* table, char** names, dmDataType* types, long* strlens, 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.

## 16.6 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.

## 16.7 dmTableGetColOffset

short dmTableGetColOffset(dmBlock *tab, long colno)

(R1.7) Returns the byte offset of the the specified column within the table row, accounting for any (platform-specific) unnamed field padding that might occur within the row structure.

Recall that colno is ones-based. See also dmTableAllocRow.

## 16.8 dmTableGetColPtr

void* dmTableGetColPtr(dmBlock *tab, void *row, long colno)

(R1.7) Like dmTableGetColOffset, but returns a pointer to the beginning of the column data within the passed row structure.

## 16.9 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).

## 16.10 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 are 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. This is fine for small tables, but in general we recommend that instead of checking GetNoRows and looping up to that value, it's better to use a while loop that checks for dmNOMOREROWS.

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

Context 3 (R1.74): Fixed to correctly (but inefficiently) return the number of rows for a filtered table.

## 16.11 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.

## 16.12 dmTableGetRowNo

long dmTableGetRowNo(dmBlock* table)

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

Context 2 (R1): Image: Returns 1.

## 16.13 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.

## 16.14 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 (R1.5): Open a virtual table datablock and its dataset. The virtual data specification is parsed according to the filtering syntax in the Data Manipulation User's Guide document. The underlying physical dataset is opened, after which filters are set up to provide the virtual view of the 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 datasets are completely independent, but at the kernel layer you may want to only open the underlying file once.

Context 3 (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.

NOTE: In R1.7 this function was renamed, from dmDatasetTableOpen.

## 16.15 dmTableOpenUpdate: Open table and dataset for update

dmBlock* dmTableOpenUpdate(char* vdataSpec)

This routine is like dmTableOpen, but if no filter is specified it opens the block read/write. We don't support read/write access through a filter.

## 16.16 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). 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): Support wild cards in the name search, like FITSIO does.

## 16.17 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.

## 16.18 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.

## 16.19 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 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).

As of R1.7, an alternative to completely definining your row structure prior to opening the table and selecting the columns is to use the dmTableAllocRow, dmTableGetColOffset and dmTableGetColPtr family of routines.

Also as of R1.7, since the DM column selection filtering functionality has been implemented, this routine is diminishing in relevance and may be rendered obsolete in forthcoming releases.

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 (R2.1): 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.

## 16.20 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.

## 16.21 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. This is likely to be flaky for filtered tables. 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.

(R1.96) Make this work cleanly for filtered tables.

### 16.22 dmTanPixToWorld

void dmTanPixToWorld( double* pix, double* asp, double* crpix, double* cdelt, double* world )

(R1.94) Convert from pixel to world coordinates. Efficient routine to avoid the use of dmCoordCalc, for the special case of a TAN projection. Input: double pix[2], the tangent plane coordinates in pixels; double asp[3], the RA and Dec of the reference point in degrees and the roll ange in degrees; double crpix[2], the reference point in tangent plane pixels; double cdelt[2], the scale in degrees per pixel (with cdelt[0] usually negative since RA decreases with X). The output is a vector with the RA and Dec.

### 16.23 dmTanWorldToPix

void dmTanWorldToPix( double* world, double* asp, double* crpix, double* cdelt, double* pix )

(R1.94) Convert from world to pixel coordinates. Efficient routine to avoid the use of dmCoordInvert, for the special case of a TAN projection. See dmTanPixToWorld for details.

## 17 New Routines

The new routines added since initial public release are listed here instead of in the main listing.

### 17.1 dmBinningParse

long dmBinningParse(dmBlock* block, char* name, char* spec, int compact, double** mins, double**maxes, long* n, int* partial)

(R1.99) Routine to parse a binning specification. In simple use, the first two arguments may be null, and the spec argument is interpreted as a DM binning specification, returning the corresponding bins in the mins, maxes arrays, and the number of bins in the variable n. The flag 'partial' is set to 1 (dmTRUE) if the last bin is a partial bin (e.g. bin 1:10:7 defines the two bins 1:7, 8:14, with the second bin being partially filled since the upper bound 10 is less than the top of the final bin).

(R1.99) In advanced use, the first two arguments represent a DM block and the name of a column or axis in the block. In this case, the default bounds for this column are used in determining defaults for the ranges. Thus, the string "pha=::4" might be equivalent to "pha=0:8192:4" if the pha column exists in the block and has the defined range 0 to 8192. If the argument 'compact' is 1 (dmTRUE), the default range is taken from the current filter (subspace); otherwise it is taken from the total valid range (TLMIN/TLMAX).

## 17.2 dmBlockCopy

int dmBlockCopy( dmBlock* src, dmBlock* dest, char* option)

(R1.99) This routine copies parts of a block. Currently supported values of 'option' are HEADER, which copies all DM header descriptors, and SUBSPACE, which copies the data subspace.

## 17.3 dmBlockCopyCol

int dmBlockCopyCol( dmBlock* dest, dmDescriptor* col )

(R1.99) Copy a column (from another block to the block described by the first argument). Only the column structure is copied, not the data. Experimental routine (meaning, be alert for problems).

## 17.4 dmCopyGeneric

int dmCopyGeneric( dmDescriptor* src, dmDescriptor* dest)

(R1.99) Copy table data for one column to another (usually in another block), for the current row only. Handles all data types and vector and array columns. However, caller must ensure that structures (data type, array size, etc) of the descriptors are identical, as no casting or checking is done. Useful for copying data after a dmBlockCreateCopy.

## 17.5 dmRegConvertWorldRegion

regRegion* dmRegConvertWorldRegion( dmDescriptor* dd, regRegion* region )

(R1.99) Force a region to be in physical coordinates if it is specified in world coordinates. If the user can supply "circle(12:26:03,+23:12:11,5')" (world coords) or "circle(4096,4096,20)" (physical coords) and in code you want to test "regInsideRegion", you don't want to have to worry about how the region is specified. We take the approach that we always use physical coords. But the parser can't know how to convert world to physical- you need to tell the region the transformation rule, which we can pass in as a coordinate descriptor (the result of dmGetCoord giving the sky-to-world transformation).

dmRegConvertWorldRegion, then, converts the region in place, changing world coordinates and radii to physical coordinates and radii using the supplied coordinate transform descriptor. If the region is already in physical coordinates, the routine has no effect. You should call this routine between dmRegParse and regInsideRegion, and then pass the physical coordinates to be tested to regInsideRegion.

## 17.6 dmRegParse

regRegion* dmRegParse( char* buf )

(R1.99) Parse a CIAO region string and return a region pointer. This routine wraps the regParse routine (see region library documentation) but also supports the

```
region(filename)
```

syntax, where filename is an ASCII region file or a CXC FITS region. Since reading the FITS region file requires the DM, this routine is included in the DM interface instead of the region library. It is recommended that dmRegParse be used instead of regParse in all applications which link to the DM.

## 17.7 dmTableGetRowSize

long dmTableGetRowSize(dmBlock* table)

(R1.99) Return the number of bytes in the table row buffer. This is the amount of storage that dmTableAllocRow would use.