CXC-DM-008

CXC Data Model

Vol. 8

C Programmers' Guide

Chandra X-ray Center

October 22, 2001

Contents

Copyright, Acknowledgement, Disclaimer

The software described in this document is freely distributed under the following copyright:

Published papers making use of CXC software should include the following acknowledgement:

> This work has made use of software provided by the Chandra X-ray Center, operated by a grant the Smithsonian Astrophysical Observatory by the National Aeronautics and Space Administration.

Contributors and 'Change Doc' Page

Jonathan McDowell, Michael S. Noble, Kenny Glotfelty, Oliver Oberdorf, Scott Randall
WWW: http://chandra.harvard.edu/

Preface

This Guide describes the SAO/CXC data model software, which allows the user to manipulate data by filtering and binning it. The CXCDM (CXC Data Model) library is used throughout the CXC software to read and write data files, and filters those data files using a special 'virtual file' syntax which qualifies the input filename. This means that users can use any of the CXC tools to filter their data on the fly, whenever an input filename is prompted for. The CXCDM also comes with some basic tools for simple data manipulation. See the Data Manipulation User's Guide for a description of the virtual file syntax and use of the tools. The present document is intended for programmers who want to write their own code using the CXCDM .

The CXCDM was developed by a team at the Chandra X-ray Center (CXC, formerly ASC), at the Smithsonian Astrophysical Observatory, Cambridge, Massachusetts, USA. CXCDM will be an integrated part of the CXC Data Analysis System, and is being distributed by the CXC as part of the flight software release in spring 1999 ready for the launch of Chandra in mid-1999.

1 Introduction

1.1 Overview

The DataModel is an I/O subroutine library which

- Gives access to different file formats
- Provides a high level, common abstraction of those formats
- Allows the application program to transparently access a filtered view of the underlying file, e.g. selecting rows and columns of tables.

1.2 FITS and QPOE

The DataModel gives you an abstract view of astronomical data files and provides data I/O transparently to FITS, QPOE and IMH format files. Instead of using format-specific calls that involve concepts specific to those formats (like the BITPIX variable in FITS files), we provide a uniform interface which deals in terms of a more abstrat description - the "Data Model". The lower layers of the library which deal with specific formats are called "kernels". The two file kernels currently supported by the DataModel architecture are the FITS kernel and the IRAF kernel. The FITS kernel provides I/O to FITS files (including images, and binary and ASCII tables, but with some limitiations, particularly for ASCII tables and variable-length array columns in binary tables). Each FITS kernel 'dataset' is a single FITS file. The IRAF kernel handles IMH image files and QPOE table/event list files. By default it currently treats a whole directory as the 'dataset', but individual files can also be datasets. The use of directories as datasets is now deprecated.

1.3 Basic concepts

The DataModel treats data as a hierarchy of **datasets**, **blocks** and **descriptors**. Loosely, datasets represent files, blocks represent tables and images (including their header information), and descriptors represent individual columns, header keywords, coordinate systems, and other named objects within a block. For instance, a table column has a descriptor, since it has a name, but a table row has neither a name nor a descriptor. The unified 'descriptor' concept helps us do useful, flexible things like treating header keywords as table columns whose value is the same in each row. Descriptors have other associated descriptive information (hence the name), such as units, comments, and data type.

- A **dataset** is an ordered set of **blocks**.
- A **block** consists of **header**, **data**, and a **data subspace** which describes the range of applicability of the data (specifically, how the data has been filtered).
- Each block consists of a set of **columns**; each column in the block has the same non-negative number of **rows**.
- In each row, the column contains a **cell** which is an n-dimensional array of **elements** (but usually this n-dimensional array is a single element, i.e. n = 0).
- Elements are vectors of **values** (but usually just one value). For example, an (X,Y) position pair is a 2-dimensional vector. In the DataModel, we distinguish between **vectors** like (X,Y,Z) (different quantities grouped together) and **arrays** like X[10] (several values of the same quantity), so that you can in the worst case have a **vector**

array like (X,Y,Z)[10]. In most cases, though, data is **scalar**, i.e. neither vectorized nor arrayed.

- A **value** can be numeric, string, or one of the other supported data types described below.
- An image block has one row and one column, containing a single N-dimensional cell of values.

There are several flavors of descriptor:

key descriptor which corresponds to a DataModel header key. In the DataModel, kernel header keys which describe the **structure** of the file are not visible through the DataModel interface. For instance, a FITS TUNITn keyword which describes the unit for a table column doesn't count as a DataModel header key - instead, you control it through altering the properties of that table column. This lets you concentrate on just the 'extra' header keys which contain scientific rather than structural information; these are the DataModel header keys.

column descriptor which corresponds to a table column. Since images are considered to be a trivial table, there is also a single 'image data descriptor' for each image block. This is currently a separate type of descriptor, but the distinction will soon be phased out. Some columns are 'vector columns' with multiple component columns, like 2-dimensional positions.

subspace descriptor which corresponds to filtering information on the block. Each quantity that the block has been filtered on has a corresponding descriptor. Sometimes there is an associated column descriptor too - maybe TIME is a column and you've also filtered on TIME - and sometimes not: maybe you filtered on PHA but then got rid of that column, so only the filter information is left.

coord descriptor which represents a 'pseudo-column' defined as a function of another column. Currently we use these to implement support for WCS (World Coordinate System) information. A special case, the physical coordinates along the axes of an image, are called **axis group** descriptors.

1.4 Virtual Files

When you open a DataModel block to read it, you pass the subroutine (e.g. dmTableOpen) a string called a 'virtual file specification' or 'vspec', rather than simply a file and table name. The block descriptor that is returned refers to that virtual file, and all I/O is done in terms of the filtered view described by it. For instance,

```
table = dmTableOpen( "bas.fits[stdevt][pha=20:30]" );
```

opens a virtual file which consists of only those rows in table 'stdevt' of file 'bas.fits' which have values of the PHA column lying between 20 and 30. Unlike some virtual file implementations, DataModel filtering does not read in the entire filtered file at open time, although some buffering is done as you read through the file. This means there's no limitation on the size of file you can read, but it's inefficient to randomly access rows of a filtered file (e.g. going to row number 42 may require the file to be filtered again).

In contrast, DataModel binning, e.g.

```
image = dmImageOpen( "bas.fits[stdevt][bin x=32,y=32]" );
```

does create the entire binned image in memory the first time you try and read from the data section of the virtual file. Binning may fail if insufficient memory is available. (We plan to add code to get around this by rebinning on subsections, but that will be an 'after launch' addition). However, you can access the header and structure of an arbitrarily large binned image without triggering the binning - it's only when you read image pixels that binning occurs.

1.5 A simple example

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

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

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

```
#include "ascdm.h"
```

```
#include <stdlib.h>
#include <stdio.h>
#define MAXPHA 256

int main( int nargs, char* args[] )
{
dmDataset* input_ds;
dmDataset* output_ds;
dmBlock* gti_table;
dmBlock* event_table;
dmBlock* out_table;
dmDescriptor *start_col, *stop_col, *pha_col;
dmDescriptor *status_col;
dmDescriptor *channel, *counts, *rate;
double* start;
double* stop;
double livetime, tzero;
long spectrum[MAXPHA+1];
long pha, itzero, i, ngti, n, row;
short status;

for( i = 0; i < MAXPHA; i++ )
 spectrum[i] = 0;


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

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

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

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

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

```
/* Look for any of various MJDREF keywords */
if( dmKeyRead_l( event_table, "MJDREFI", &itzero ) )
{
   (void)dmKeyRead_d( event_table, "MJDREFF", &tzero );
    tzero += itzero;
}
else if( !dmKeyRead_d( event_table, "MJDREF", &tzero ) )
   if ( dmKeyRead_l( event_table, "XS-MJDRD", &itzero ) )
   {
      (void)dmKeyRead_d( event_table, "XS-MJDRF", &tzero );
       tzero += itzero;
   }
   else
   {
       tzero = 0.0;
       printf( "No MJDREF keyword found\n" );
   }

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

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

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

```
dmKeyWrite_c(out_table, "CHANTYPE", "PHA", " ", "PH binning type");

for (pha = 0; pha <= MAXPHA; pha++)
{
    dmSetScalar_l(channel, pha);
    dmSetScalar_l(counts, spectrum[pha]);
    dmSetScalar_d(rate, spectrum[pha]/livetime);
    dmTablePutRow(out_table,NULL);
}
dmBlockClose( out_table );
dmDatasetClose( output_ds );
return 0;
}
```

The same program can be written using explicit row data buffers and one-block-at-a-time dataset handling (some details omitted where the same as the previous version). The one-block-at-a-time dmTableCreate/dmImageCreate and dmTableOpen/dmImageOpen routines are a convenience for the special case of a dataset with one block in it, which is very common. It minimizes the number of handles floating around in the program. The row buffers have the advantage of simplicity, but the huge disadvantage that the code is no longer robust to changes in data type and column order in the input file. Therefore, it should be used with caution in production code intended for wide use.

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


gti_table = dmTableOpen( "bas.fits[STDGTI][cols START,STOP]" );
livetime = 0.0;

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

```
dmTableClose(gti_table);

event_table=dmTableOpen("bas.fits[STDEVT][columns PHA,STATUS]");

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

dmTableClose(event_table);

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

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

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

dmTableClose( out_table );
```

Better yet, you can also rewrite the first part of the code as follows:

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

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

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

dmTableClose( event_table );
```

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

1.6 Online CXC DataModel References

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

2 Programming Considerations

2.1 Configuration and Sample Code

A configure script lets you include or exclude specific kernels and configure the library for a particular platform.

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

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

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

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

DataModel programs link to the following libraries:

- libascdm.a, the main DataModel library
- libwcs.a, Doug Mink's coordinate transformation library
- libregion.a, the DataModel Region library for 2D region filtering.
- libcfitsio.a, Bill Pence's CFITSIO library (if the FITS kernel is enabled)
- libirafm.a, the CXC repackaging of the IRAF libraries (if the IRAF kernel is enabled)
- On some systems, libl.a and liby.a (the LEX and YACC libraries, or their FLEX and BISON equivalents) may need to be explicitly linked, as well as the libm.a C math library.

2.2 Structure Objects

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

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

2.3 Memory Management

To a large extent the CXC DataModel does not require the user to worry about details of memory management. For example, it is not necessary to free memory associated with individual dmDescriptor, dmBlock, or dmDataset pointers. Memory allocated to blocks and the descriptors they may contain will be freed when block is closed via an appropriate routine call. Similarly, memory allocated to dmDataset pointers will be freed when the dataset is properly closed. To summarize, you should remember to close blocks and datasets, but you never need to explicitly close a descriptor.

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

This maximum length does not include the C null termination, but the DM will enforce null termination of returned strings. Thus, in the call

```
dmGetScalar_c( dd, value, maxlen )
```

the variable "value" must be declared char[maxlen+1], and the memory location value[maxlen] may be set to zero.

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

2.4 Defined Types

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

2.5 dmDataType

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

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

The dmSHORT class is a machine-dependent #define to a 2 byte integer type; on many machines short will be equivalent. Same goes for other types. The logical and blockref types are typdef'd not #defined to make sure they are compiler-distinct from integer and string. Other types may be added later. The blockref type is intended for use as a special 'URL/file reference' type, but we haven't fully implemented it yet.

2.6 dmBlockType

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

dmBlockType	String
dmTABLE	TABLE
dmIMAGE	IMAGE
dmUNKNOWNBLOCK	UNKNOWN

2.7 dmDescriptorType

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

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

2.8 dmElementType

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

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

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

3 Interface Parameters

3.1 Kernel Mnemonics

The DM kernel mnemonics provide an encapsulated way of referencing the the ETOOLS kernel currently being used for I/O on a given dataset. Note that multiple datasets may be opened during the execution lifetime of a DM application, with a potentially distinct kernel used for I/O on each.

Kernel Mnemonic	Value	Description
dmFITSKERNEL	”FITS”	interface to CFITSIO
dmIRAFKERNEL	”IRAF”	interface to native IRAF

3.2 Kernel Options

The kernel options, invoked with dmKernelSetOption, allow you to fine-tune the behaviour of the kernels outside of the DataModel paradigm. For instance, the DM doesn't know the difference between FITS BINTABLE and FITS ASCII Table representations, so we have to control them by the 'back door'. The TABLE=STSDAS and COL=VARARRAY options are not yet supported.

Kernel Option	Effect
'TABLE=BINTABLE'	Write tables in BINTABLE format if kernel is FITS
'TABLE=ASCII'	Write tables in ASCII table format if kernel is FITS
'COL=VARARRAY'	Next created array col is a variable array, if kernel is FITS
'TABLE=QPEVT'	Write tables in QPOE event format if kernel is IRAF
'TABLE=STSDAS'	Write tables in STSDAS format if kernel is IRAF

3.3 Internals

The DM internal parameters are used by the dmSetInternals routine.

Internal Parameter	Value	Effect
dmTABLEBUFFERSIZE	"buffersize"	Tune the size of the internal DM table buffer. Note that you must not set the buffer size smaller than the largest number of rows that your program will read in any single table I/O call. Setting this parameter to a low number can reduce the amount of memory your program consumes, or conversely, setting it higher may increase efficiency by ensuring that large tables are handled by larger buffers.

3.4 Error Handling and Diagnostics

Most of the CXC DataModel routines indicate completion status either by returning an "int" status code or by returning unusual values (e.g., NULL pointers or negative row numbers). The #define symbol "dmErrCode", equivalent to "int", is also provided for use with the return status codes. Regardless of whether or not a DataModel API function provides explicit error state indication, the call completion status can be determined by using the **dmGetError** and **dmGetErrorMessage** routines.

The **dmFAILURE** status code indicates some error condition exists, while **dmSUCCESS** indicates the call completed successfully. Other status codes and return values are listed as appropriate with the associated APi functions. The numeric values may change in the future, and we are considering various schemes such as using negative values to indicate a non-fatal error or warning.

Use **dmGetVersion** to find the current DataModel release version. This may be needed if you send email to the CXC about possible bugs.

In later releases, we hope to provide **dmDatasetPrintKernel** and **dmBlockPrintKernel** to inspect the contents of the file at the kernel level, bypassing the layer of interpretation imposed by the DataModel. Use **dmBlockGetNoKernelKeys** and **dmBlockGetKernelKey** to inspect header entries at the kernel level. Until these routines exist, you should use the appropriate native tools (FTOOLS fdump, PROS qplist) to get a kernel-level view of the files.

3.5 Counting in the DataModel

The CXC DataModel uses a ones-based counting system consistently. That is, the smallest block number, key number, image pixel coordinate, column number, or row number will ALWAYS be 1. In particular, note that FTOOLS counts FITS HDUs from zero, while we count from 1.

3.6 Initialization Routines

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

Users wishing to explicitly perform initialization at some well-defined point within their application may use the dmInit routine.

3.7 Multithreading

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

4 Introduction to the DataModel library routines

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

4.1 Dataset operations

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

4.1.1 Opening and closing files

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

Table 1: Open/close routines

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

dmTableOpenUpdate	Open table and dataset at same time, read-write, return dmBlock*
dmTableCreate	Same for creation
dmTableClose	Close dataset opened by dmDatasetTableOpen/Create
dmImageOpen	Open image and dataset at same time, return dmBlock*
dmImageOpenUpdate	Open image and dataset at same time, read-write, return dmBlock*
dmImageCreate	Same for creation
dmImageClose	Close image

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

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

4.1.2 Navigating within a dataset

Most of the time we work with a single block within the dataset. If you have an open dataset, and want to change to a different block within the dataset, how do you get there? There are several ways. You may access the blocks sequentially, or by number, or by name.

To access the blocks sequentially, use the **dmDatasetNextBlock** routine. This routine will open the next block, (the first time, it will open the first block in the dataset), and repeated calls will go through all the blocks until the end of the dataset, when it will return null. The **dmDatasetGetCurrentBlockNo** inquiry routine returns the number of the most recent block to have been opened. The **dmDatasetAdvanceBlocks** routine moves ahead or back by a specified number of blocks, so dmDatasetNextBlock is equivalent to calling dmDatasetAdvanceBlocks with an argument of 1.

To access the blocks by number, use **dmDatasetMoveToBlock**. **dmDatasetGetBlockName** lets you check the name of a numbered block before opening it. The name of the dataset itself is available via **dmDatasetGetName**.

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

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

4.1.3 Kernel related routines

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

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

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

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

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

4.1.4 Auxiliary dataset routines

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

- **dmFileExists** is used to test existence of a file.
- **dmDatasetAccess** is used to test whether a dataset exists, prior to opening it. It actually opens the dataset, and is more general than dmFileExists since it handles the case where a dataset consists of multiple files, but will fail if the file is corrupted.
- **dmDatasetDestroy** deletes a dataset on disk by name.
- **dmDatasetDelete** deletes a dataset on disk which is already open; often it's more robust to use dmDatasetDestroy instead.

4.2 Tables

4.2.1 Opening a table

You can open an existing table in the following ways:

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

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

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

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

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

4.2.2 Basic table properties

.

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

4.2.3 Creating table structure

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

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

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

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

Another kind of grouped column imposes a particular meaning on the grouping, using the concept of compound element types (also known as Intervals). This allows us to store ranges of values rather than point values. For instance, the Good Time Intervals (START,STOP) are more elegantly handled as a RANGE element for the TIME variable. Columns of this kind are created with the **dmColumnCreateElement** routine (not yet supported).

Combining the high level constructs to produce arrays of vectored compound element types may be done using the (not yet supported) **dmColumnCreateGeneric** routine; all the other column create routines are special cases of this. You can define a set of columns at once with **dmTableCreateColumns** (scalar columns only) or **dmTableCreateGenericColumns** (generic columns).

4.2.4 Navigating in the table

To start with, you are always at the first row of the table. Repeated read/write operations will not change the row. To move to another row, use **dmTableNextRow** or **dmTableSetRow** (but to write a row, you must use **dmTablePutRow**, as described below). To find out which row you are at, use **dmTableGetRowNo**. The routine **dmTableGetNoRows** tells you how many rows are in the table. However, note that in a filtered table, **dmTableGetNoRows** has to filter the entire table to figure this out, so if you can avoid it, do.

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

4.2.5 Cell-based I/O: introduction

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

4.2.6 Column properties

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

- **dmGetName**, **dmSetName** - get/set name of column
- **dmGetUnit**, **dmSetUnit** - get/set unit of column
- **dmGetDataType** - get data type of column (cannot be changed)
- **dmGetDesc**, **dmSetDesc** - get/set descriptive comment for column
- **dmDescriptorGetLength** - get length of variable (bytes for string, bits for bitmask, 0 otherwise).
- **dmGetArrayDim** - get array dimensionality for column (cannot be changed)
- **dmGetElementDim** - get vector dimension for column (cannot be changed)
- **dmGetElementType** - get element type of column (cannot be changed)
- **dmGetDisp**, **dmSetDisp** - get display format hint for column
- **dmColumnGetNo** - get number of column in table
- **dmDescriptorGetRange**, **dmDescriptorSetRange** - get/set legal range of values for column.
- **dmDescriptorGetBin**,**dmDescriptorSetBin** - get/set default binning factor for column
- **dmDescriptorGetNull**,**dmDescriptorSetNull** - get/set null value for column

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

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

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

To find out which block your descriptor belongs to, use **dmDescriptorGetBlock**.

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

4.2.7 Cell-based I/O read and write

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

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

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

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

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

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

```
dmTableNextRow( table );
```

4.2.8 Cell-based I/O: complicated cases

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

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

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

4.2.9 Column-based I/O

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

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

4.2.10 Row-based I/O

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

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

If this structure corresponds exactly to the table structure, you can directly use the dmTableGetRow routines to fill `myrow` with the data and use e.g. myrow.status as a variable. If the table might have extra rows or have the rows in a different order, you have to tell the dm library explicity what your myrow struct contains. To do this, use the "[cols pha,status,time]" selection operator of the virtual file syntax.

Note: for string columns, include the extra byte for the terminating null of a C string. For example

```
struct { long pha; char label[STRSIZE+1]; dmBool flag; double values[7] } myrow;
```

See the program examples/dmtest.c in the datamodel source tree for code which uses such a row.

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

The advantages of row-based I/O are balanced by the disadvantage of a lack of type checking on the data. Also, if you don't know what data is in the table in advance (generic table browsing or calculation tools), things get a bit trickier. The **dmTableAllocRow** routine, and the **dmTableGetColOffset** and **dmTableGetColPtr** routines, provide further row-based I/O functionality to support run-time definition of the row-based I/O structure.

4.2.11 Preferred Axes

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

The **dmBlockSetPref** may be used to record the defaults in the table; The **dmBlockGetPref** routine may then be used to extract the information.

4.3 Coordinate Descriptors

4.3.1 Coordinates

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

To write a coordinate system on a table column or an image axis group, use the **dmCoordCreate** routines. To make an image axis group (a 'physical coordinate system' in IRAF terminology), use **dmArrayCreateAxisGroup**. To get the group number of the axis group use **dmCoordGetAxisGroupNo**, and to open an axis group use **dmArrayGetAxisGroup**.

To find the default coordinate associated with a descriptor (if any), call **dmDescriptorGetCoord**. There may be more than one coordinate associated with a descriptor; **dmDescriptorGetNoCoords** and **dmDescriptorGetCoordNo** may be used to get them all. Conversely, **dmCoordGetParent** may be used to find the parent descriptor of a coordinate descriptor.

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

4.3.2 Coord values

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

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

Example:

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

4.3.3 Physical and world coordinate systems

Images have both physical and world coordinate systems. (Tables don't have physical systems; the column values are considered to be the physical values). The Image LOGICAL COORDINATES are just the pixel numbers. In the DM, we imagine that for each logical axis, there is a physical axis which has a linear scaling on the logical axis, and there may also be a world coordinate axis which is a further transform on the physical axis.

Remember that a 2-D image in the DM can consist either of two axis groups each with one subaxis, or of a single axis group with two subaxes. For instance, an image with an RA,DEC WCS has a single axis group (NGROUPS = 2) and the group has dimension 2. The coordinate systems attach to the groups, not the individual axes, so there is a single physical coord descriptor and a single world coord descriptor in this case, instead of two separate ones for each axis. That's because the mapping of X and Y to RA and DEC mixes X and Y inextricably. You can yse **dmArrayGetNoAxisGroups** to get the number of axis groups in an image.

Examples:

```
         Logical        Physical         World


Quantity  Binned pixel   Original pixel   RA, Dec value
Name      (X_BIN,Y_BIN)  SKY(X,Y)         EQPOS(RA,DEC)
Type      Always integral  Floating?      Floating
Unit      -              pixel            deg
```

```
Quantity  Light curve bin  Mission time   Julian day
Name      TIME_BIN        TIME           JD
Unit      pixel           s              d
Type      integral        double         double
```

To read these from a 2D image, we do:

```
  dmDescriptor* imageData;
  dmDescriptor* phys[2];
  dmDescriptor* world[2];
  long ngroups, group, subaxis, axis, dim;
  double pcrpix[2], pcrval[2], pcdlt[2];
  double wcrpix[2], wcrval[2], wcdlt[2];

  ngroups = dmArrayGetNoAxisGroups( imageData );
  for ( group = 0; group < ngroups; group++ ) {
   phys[group] = dmArrayGetAxisGroup( imageData, group+1 ); /* 1-based group no*/
   world[group] = dmDescriptorGetCoord( phys[group] );
   dim = dmGetElementDim( phys[group] );
   dmCoordGetTransform_d( phys[group], pcrpix, pcrval, pcdlt, dim );
   if ( world[group] != NULL ) {
    dmCoordGetTransform_d( world[i], wcrpix, wcrval, wcdlt, dim );
    for ( subaxis = 0; subaxis < dim; subaxis++ ) {
     axis = group + subaxis;
     logical_to_world_pixel_size[axis] = pcdlt[subaxis] * wcdlt[ subaxis ];
    }
   }
  }
```

Note that in the FITS file, the logical-to-world transform is stored in the CRPIX/CRVAL/CDELT keywords and the logical-to-physical transform is stored in the C1RPX/C1RVL/C1DLT keywords. The data model combines these to return the physical-to-world transform and the logical-to-physical, so you have to do a bit more work to get the logical-to-world information.

To make an image with these,

```
  char* pname = "SKY";
  char* punit = "pixel";
```

```
char* pcptNames[] = "X", "Y";
long dim = 2;
double pcrpix[2] = { 128.0, 128.0 };
double pcrval[2] = { 256.0, 256.0 };
double pcdelt[2] = { 2.0, 2.0 };
char* wname = "EQPOS";
char* wunit = "deg";
char* wcptNames[2] = "RA", "DEC";
char* wtransform = "TAN";
double wcrpix[2] = { 256.0, 256.0 };  / * Identical with pcrval */
double wcrval[2] = { 271.3, -30.21 };  /* Corresponding RA and Dec */
double wcdelt[2] = { -0.0032, 0.0032 };

/* Create linear logical-to-physical transform with initial value the identity transform */
phys[i] = dmArrayCreateAxisGroup( imageData, pname, punit, pcptNames, dim );
/* Adjust value of transform parameters */
dmCoordSetTransform_d( phys[i], pcrpix, pcrval, pcdelt, dim );
/* Create physical-to-world transform */
world[i] = dmCoordCreate_d( phys[i], wname, wunit, wcptNames, dim, wtransform,
           wcrpix, wcrval, wcdelt, NULL );
```

An example of the use of the physical coord system: suppose you want to find the off axis angle of a target pixel in a rebinned sky image, given that you know the mean aspect. In the CXC analysis system, the rebinned sky image's physical coords would be the sky pixel coords. The definition of sky coords is that the tangent point corresponds to the nominal pointing direction; in the absence of aspect info that is a good first guess. If you have the RA_PNT, DEC_PNT keywords that will give you the RA and Dec of the mean pointing. We can use CoordInvert to map these to physical coords, and use CoordCalc to map your target logical pixel to physical coords.

```
dmKeyRead_d( imageData, "RA_PNT", &optax_eq[0] );
dmKeyRead_d( imageData, "DEC_PNT", &optax_eq[1] );
dmCoordCalc_d( phys[0], target_pixel, target_phys );
dmCoordInvert_d( world[0], optax_eq, optax_phys );
distance_in_phys_pixels = root_add_squares( optax_phys[0]-target_phys[0], optax_phys[1] - target_phys[1] );
distance_in_arcsec = distance_in_phys_pixels * wcdlt[1] * 3600.0;
```

4.3.4 Coord properties

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

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

(Some of these don't do anything useful yet in the case of coordinates).

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

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

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

4.4 Header keys

4.4.1 Header keys

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

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

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

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

You should therefore be aware that in future key reads may need to take into account the element and array dimension of the keys.

To find the total number of keys in the block, use **dmBlockGetNoKeys**.

To read a header key from a block, you have the following choices:

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

4.4.2 Key properties

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

- **dmGetName**, **dmSetName** - get/set name of key

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

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

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

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

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

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

4.4.3 Comments

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

4.5 Images

4.5.1 Opening an image

You can open an existing image in the following ways:

- Open the next block in a dataset with **dmDatasetNextBlock**
- Open a numbered block in a dataset with **dmDatasetMoveToBlock**.
- Open a block by name with **dmBlockOpen**
- Open a block and a dataset at the same time using **dmImageOpen**

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

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

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

4.5.2 Basic image properties

.

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

To find the dimensionality of the image, the **dmGetArrayDimensions** routine tells you what and how long each axis is.

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

Example:

```
long* axes;
dmBlock* image = dmImageOpen("image.dat");
char name[MAXLEN];
dmBlockGetName(image,name,MAXLEN);
dmDescriptor* data = dmImageGetDataDescriptor(image);
dmDataType type = dmGetDataType(data);
naxes = dmGetArrayDimensions(data, &axes);
free(axes);
dmImageClose(image);
```

4.5.3 Image axes

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

Example:

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

4.5.4 Image data

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

4.5.5 Image properties

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

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

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

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

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

4.5.6 Image pixel lists

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

4.6 Data Subspace

4.6.1 Subspace columns

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

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

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

We might instead write

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

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

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

The SubspaceColCreate code will recognize TIME as a special case and store the array of values in a separate GTI table. You can also force data to be stored in a separate table in FITS using SubspaceColCreateTable; other kernels may do something different, but it should be analogous to whatever they do for GTIs.

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

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

4.6.2 Subspace column properties

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

- **dmGetName**, **dmSetName** - get/set name of subspace descriptor
- **dmGetUnit**, **dmSetUnit** - get/set unit of subspace descriptor

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

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

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

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

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

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

4.6.3 Accessing subspace columns

- **dmBlockGetNoSubspaceCols** returns the total number of filters.
- **dmBlockGetNoSubspaceCpts** returns the number of separate components in the subspace (see the abstract design document for details).

- **dmBlockGetSubspaceColNo** gets a filter by number.
- **dmBlockGetSubspace** returns the full list of descriptors for the filters in the subspace.
- **dmBlockSetSubspaceCpt** sets the value of the subspace component number, used by dmSubspaceColCreate etc.
- **dmBlockGetCurrSubspaceCpt** returns the current subspace component, used by dmSubspaceColUpdate, etc.
- **dmSubspaceColGetTableName** returns the name of any associated table used by the column.

4.6.4 Subspace routines

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

- **dmBlockIntersectSubspace** creates a new data subspace which is the intersection of two others.
- **dmBlockMergeSubspace** creates a new subspace which is the union (logical OR) of two others.
- **dmBlockPrintSubspace** is a diagnostic routine to show the current values in the data subspace.