Note: This document is same as the document at nom.tam.fits site except that examples are now in C#

Using the CSharp FITS utilities.

This document describes version 1.1 of the CSharpFITS class library for reading and writing FITS files in C#. For a discussion of FITS see documentation at http://fits.gsfc.nasa.gov. FITS, the Flexible Image Transport System, is a file format defined to facilitate the exchange of scientific information. This class library provides facilities to convert FITS information to forms usable directly within C# programs, and to write information generated in a program to a FITS file.

The underlying philosophy for this library is to make simple things easy but leave hard things possible. The library encompasses quite a few classes, but users will typically need to use only a small fraction of the public interface of the system.

This document provides a quick overview of the packages and classes in the system and then discusses how to use the library using a series of simple examples. Special topics are discussed at relevant points.
Detail documentation can be found in the html documentation available with the release.

Overview of the System

The FITS classes comprise the classes in the nom.tam.fits package and make extensive use of the nom.tam.util package. The nom.tam.image package provides support for dynamic image sub setting.
The current release of the package is updated to .NET framework version 2.0. In particular the GZip compression feature for Fits file
Depends upon methods that were defined in .NET 2.0 framework version

nom.tam.util:

BufferedDataStream
BufferedFile
ArrayDataIO
RandomAccess

These classes are used to provide efficient access to large amounts of data. Basically they allow arrays to be read and written efficiently. The Stream classes provide input and output for streams, while the `BufferedFile` class provides efficient access for uncompressed, local data using an underlying `RandomAccessFile`. The standard classes are slowed by frequent synchronization and the many method invocations required reading large arrays.

The `ArrayDataIO class` is an interfaces that is implemented as appropriate by the `BufferedXXX` classes. The `RandomAccess` interface extends `ArrayDataIO`. It is implemented only by `BufferedFile` currently.

These classes are general utilities and can be used completely independently of the FITS library.

ByteParser
ByteFormatter

These two classes provide more efficient translations between ASCII and binary representations of numbers than the standard FITS classes. Note that for real numbers these classes may suffer small round-off errors compared to the standard classes.

These classes may be useful outside of the FITS context but so far are used only in FITS ASCII table classes.

ColumnTable
DataTable

These classes are used within FITS binary tables. The `ColumnTable` class provides a mechanism for efficient access to non-homogeneous data. They also allow a Binary table to be created using relatively few objects (essentially one object per table) rather than one object per entry. Both of these dramatically enhance the useability of binary tables. The `DataTable` interface defines how a `ColumnTable` is to be accessed but it not used in the FITS libraries.

The ColumnTable should be usable outside of the FITS library whenever a table of heterogeneous data is to be read. Homogeneous data (i.e., data composed of a single primitive type) is more likely to be easily read with classes that implement the ArrayDataIO interface.

`ArrayFuncs`

The `ArrayFuncs` class defines a large number of static utility functions for manipulating arrays, especially primitive arrays. Most handle multi-dimensional arrays transparently. Facilities include:

- Generating deep clones of arrays.
- Copying an array to an array of another type.
- Determining the total size of an array.
- Determining the base type of an array.
- Converting a multi-dimensional array to one-dimension (flattening).
- Converting a one-dimensional array to multiple dimensions (curling).
- Examining an array.
- Extracting the shape of an array.

`HashedList`
Cursor

The `HashedList` is a `Collection` defined to support FITS headers. Basically it provides support for a list where elements may optionally have keys. The inner class `HashedList.HashedListIterator` implements the Cursor interface and is extremely helpful in manipulating FITS headers. The Cursor interface extends the System.Collections.IEnumerator interface .It allows insertions and keyed access. The `Header.iterator()` method returns a Cursor that the user can use to view the header.

`nom.tam.image:`

`ImageTiler`

The `ImageTiler` class allows users to extract subimages from a FITS primary image or image extension.

nom.tam.fits:

Fits

The FITS class is primarily concerned with establishing a connection between the FITS object and outside world. A wide variety of constructors allow the user to read existing FITS data. A null FITS object can be created to which HDU's can later be attached.

A tiny note on capitalization: In this FITS library, acronyms that are pronounced as words, e.g., FITS and ASCII, are treated as normal words in class and variable names. Acronyms that are spelled out, e.g., HDU, are always capitalized.

XxxHDU/XxxData

Each of the types of FITS data uses a pair of classes XxxHDU and XxxData (or XxxTable). The HDU class provides the links between the header and data sections of an HDU while the Data class provides the detailed analysis and access to the underlying FITS data. Each FITS type also has an associated data kernel, a non-FITS structure in which data is actually held. For binary tables this is a ColumnTable but it is some variety of array for all other types.

ImageHDU/ImageData

These classes now include the functionality of the old PrimaryHDU and PrimaryData classes. Users can either retrieve FITS data as a multi-dimensional array or use an ImageTiler to get sub-images of the array as a one-dimensional array. Reading of image data is typically deferred until the user requests data. **Extension:** C# bytes are supported as images with BITPIX=8.

RandomGroupsHDU/RandomGroupsData

These classes support FITS random groups. Random groups are permitted in Image extensions as well as in the primary array.

Random groups data is supported as a Object[nrow][2] array. The first element for each array is the object parameter information – which may have 0 length. The second element is the data array. **Extensions**: Random groups support BITPIX=64 longs. Random groups are supported in Image extensions.

AsciiTableHDU/AsciiTable

These classes support FITS ASCII tables. The ASCII table kernel is an Object[] array. Each element of this array is a primitive double, float, int, or long array, or a String array. The constituent arrays must all have the same size. Users can construct an ASCII table dynamically by adding columns starting from a null array or by using the appropriate constructors. Note that the classes currently ignore the number of decimal places specified in the TFORM entry for real numbers. Only the length information is used in formatting data. Null fields are fully supported.

ASCII table data is not normally read until the user requests it. Users may request data by element, row, or column. The last of these will read in the entire table, but the first two will read only the requested data. The AsciiTable.getData() (or getKernel) method can also be used to ensure that all data is read.

All ASCII tables can be represented as binary tables and users can enable or disable binary tables by calling the setUseAsciiTables method in FitsFactory. If ASCII tables are enabled they will be used where possible.

Note that ASCII tables support long integers but this is not an extension of the FITS standard. Indeed FITS ASCII tables can in principal store real and floating point numbers of arbitrary length and precision which may not be represented using any of the standard C# types. This is unlikely to be a problem in practice though it is not inconceivable that there are FITS files with 8 byte integer or 16 byte real data encoded in ASCII tables.

BinaryTableHDU/BinaryTable/FitsHeap

These classes support FITS binary tables. Variable length columns are supported. Variable length column elements are returned with appropriate lengths and may be returned as zero-length arrays. The data kernel is a ColumnTable object and row and element reads are possible without requiring the kernel to be instantiated.
Extension: Long integers are supported in binary tables using the format character (in TFORM) 'K'.

UnknownHDU/UnknownData

These classes support FITS data where the internal structure of the FITS information is not known or currently supported. Data is stored internally as a byte array. You can actually create an HDU of this type to buffer conglomerations of primitive data types. Most commonly these types can be used to read standard formats in an installation of the FITS library where not all formats are supported.

FitsFactory

The FitsFactory class is used to find the appropriate FITS type. It allows users to create FITS data elements given a Header, or an HDU given a data element. When adding a new type of data to be handled in the FITS library, only the FitsFactory and the classes directly supporting the new type should need to be modified. The FITS classes now support all the accepted protocols so further extensions may be rare. Users can get most of the functionality of the FitsFactory class using convenience methods in the Fits class.

FitsUtil

This class comprises a few utilities that are needed in various elements of the FITS classes. Users should not typically need to access this class directly.

FitsDate

This class provides for translations between FITS and CSharp representations of dates. Both the old and new FITS date formats may be read.

Deferred Input.

Most FITS classes support deferred input for FITS data. If a FITS HDU is read from a non-compressed local file, then the header is read but the data section is skipped until the user requests information from it. If the user requests an entire image or table it will be read, but users may choose to read only sections of the data as appropriate for the particular type, e.g., a subset of an image or a row of a table. In this case the entire data element need never fully present in memory. Once the entire data section is read into memory, operations to read in sections of the data are still supported but work from the in-memory version rather than from the input file.

While deferred input should normally be invisible to users, it is possible to cause problems if the user mangles the FITS input stream between the time the HDU is initially scanned and when the user eventually reads the file. Note also that users must provide any synchronization needed to manage multiple accesses to a given FITS resource.

Rewriting and rereading data.

All FITS types support re-writing if the data is being read from an uncompressed local file. The system attempts to assure that the size of an HDU element has not changed when a re-write is requested. Note that elements are always a multiple of 2880 bytes, so there is some flexibility here.

Examples

The examples below sketch out how a user might perform certain functions. For maximum clarity no error checking code is shown. Examples of most of the useful calls in the FITS library are found under tests package.

Read the primary image:

```
Fits f = new Fits("filename");
ImageHDU h= (ImageHDU) f.ReadHDU();

float[][] img = (float[][]) h.Kernel;
```

While this is simple enough, note the ugly coercions required to get data. I have not been able to find any way of getting around these. Also note that this code assume you know the type of the data. If not you might want to use ArrayFuncs to parse the object returned.

No scaling:

One important thing to note about the FITS classes is that they never automatically scale data for you. You get the data exactly as it was stored in the FITS classes.

Get a subset without reading the entire image:

```
Fits f = new Fits("filename");
ImageHDU h= (ImageHDU) f.ReadHDU();
ImageTiler t = h.Tiler;
float[] tile = new float[50 * 50];
t.GetTile(tile, new int[] { 200, 200 }, new
int[] { 50, 50 })
```

This gets a 50x50 tile with lower left corner (in FITS directions) at 200,200. Note the use of the immediate array declarations. They are ugly but convenient. Remember that one can declare arrays like:

```
new int[]{x,y}
```

where x and y are variables. You can get as many subsets as you like. If you ever call h. Kernel the image will be read into memory and then subsets will be derived from the in-memory region. This might be nice if you want to do something like an animation.

If you need to get a lot of tiles it may be inefficient to create a new array for each tile. The getTile method is overloaded to allow the user to supply the input array. If this version is called, then they user may request a tile which is not fully (or even partially) contained within the image. Pixels that are not available will be left unchanged.

Create a FITS file from an image:

```
double[][] x = ….

Fits f = new Fits();
BasicHDU h = FitsFactory.HDUFactory(x);
```

```
f.AddHDU(h);
BufferedDataStream f = new
BufferedDataStream(new FileStream("Outputfile",
FileMode.Create);
f.Write(s);
```

There are also makeHDU methods in the Fits to bypass calling the FitsFactory directly. I.e.,

```
BasicHDU h = Fits.makeHDU(x);
```

These create an HDU but do not add it to a Fits object.

Read an entire FITS file and get a summary of its contents

```
Fits f = new Fits("Filename");
BasicHDU[] hdus = f.Read();

for (int i=0; i<hdus.length; i += 1) {
    hdus[i].info();
}
```

Note that this won't do anything if there's a problem at the end of the file because it will crash before it starts writing. It's a little safer to say

```
do {
      BasicHDU h = f.ReadHDU();
      if (h != null) {
           h.Info();
      }
} while (h != null) ;
```

Then you'll get information on any HDU's at the beginning of the file even if the end is corrupt.

Note, when you call readHDU the Fits object remembers the HDU it read, so that you can go back to it if you want to.

E.g., after doing the above we could have

```
BasicHDU h = f.getHDU(0);
```

to get the primary array.

Read random groups data:

```
        Fits f = new Fits("RandomGroupsFile");
        RandomGroupsHDU h = (RandomGroupsHDU)
f.ReadHDU();
        Object[][] data =
(Object[][])h.getData().getData();

        for (int i=0; i<data.length; i += 1) {
                int[] params = (int[]) data[i][0];
                int[][] img  = (int[][])
data[i][1];
                … Process a group …
        }
```

More ugly casting I'm afraid. Remember that you need to know the type to cast to. This is the cost of a statically typed language. You can create a random groups in the same way as for an image. Just make sure that your data is an appropriately formatted Object[][] array.

Currently random groups don't support deferred input.

Manipulate the FITS header

The recommended approach for this much changed in this release. We can get the Header from an HDU:

```
Header hdr = someHDU.Header;
```

There are methods in Header to access it, e.g.:

```
if (hdr.GetIntValue("TLMIN3")) {
      …
}
```

and you can add a card to the Header with

```
hdr.addValue("TLMIN4", 36, "This is a
comment");
```

Note that this will replace the old value of TLMIN4 if it existed.

The new recommended approach to managing the Header is to use a Cursor to manipulate a new Collection, a HashedList. A HashedList is just a linked list where some of the elements can be accessed using a hash. The FITS header is stored in a HashedList. A Cursor is a kind of super-Iterator to manipulate this list.

Here's how it works:

```
Cursor iter = hdr.GetCursor();
while (iter.MoveNext()) {
       HeaderCard hc = (HeaderCard)
iter.MoveNext();
       String key = hc. Key;
       …
}
```

A Cursor can be moved to any point in the Header and it can also be used to add cards to the header. E.g., suppose we want add TCX1, TCX2, and TCX3 keywords after the TCX keyword, and delete any comments we find immediately after the TCX keyword.

```
iter. Key="TCX";    // Move to just before this
keyword
 // Now after TCX
    iter.add("TCX1",
       new HeaderCard("TCX1", someValue,
someComment);

    iter.add("TCX2",
       new HeaderCard("TCX2", someValue,
someComment);
    iter.add("TCX3",
```

```
        new HeaderCard("TCX3", someValue,
someComment);

         String key = iter.MoveNext().getKey();
// Get the key
         if (key != null && key == "COMMENT") {
             iter.remove();    // Remove the
comment

}
```

This is a little longer than our previous examples, but let's see what we did. First we moved the cursor to point just before a particular keyword. If the keyword were not in the header we'd point to the end of the header. Now we wanted to add cards after this keyword, so we got it and then added three new cards.
Note how were able to add as many keywords as we wanted. Also, note that the new cards are added just before the current position of the cursor, so the add calls don't affect what is returned by iter.MoveNext(). Then we checked the subsequent cards to see if there were immediately following comments. If so we deleted them. We can only delete one entry per call to Next().

A few things to note. There are two kinds of keys here. HashedList keys are specified by the user as the first argument in the two argument add call. They'll also get set when we read key=value cards in an existing header. These keys must be unique. Each HeaderCard can also have a key. For a key=value card this should normally be the same as the HashedList key, but for other cards this might be 'COMMENT' or 'HISTORY'. In principle one could create HashedList keys to COMMENT or HISTORY cards, but the user needs to be careful not to create duplicate keys. If we try to add a keyed entry to the HashedList, it will delete an existing element with the same key.

One can have many Cursors on the same Header, which may be useful if different objects are looking at different parts of the header, but the user can get in trouble with incautious deletions.

When an HDU is created from a data object, all required structural keywords will be inserted into the Header. A user modifies those at their own risk.

When a Header is written the initial keywords are checked to see if they are legal, but this check is fairly minimal.

The Header.PositionAfterIndex method is useful when you want to add information about something in the header that takes an index (e.g., TLMIN, or CRVAL).

HIERARCH key values:

A class HierarchCard extends the HeaderCard. To add a HierarchCard one just does

```
String card = "HIERARCH TEST1 TEST2 INT=
123 / Comment                                    ";

hc = new HeaderCard(card);
iter.addKey(key, hc);

Method HierarchCard(String) in HeaderCard class is
used to process HIERARCH style cards.
The keyword for the card will be "HIERARCH. TEST1.
TEST1..."
The value will be the first token which starts with
a non-alphabetic character (i.e., not A-Z or _).
A '/' is assumed to start a comment.
```

Read an image line-by-line:

Here's something a little funkier where we use the lower level methods in the classes. We'll read in an image one-line at a time. You can judge whether
it is a good thing to support a style like this.

```
BufferedFile bf = new
BufferedFile("Input.fits", FileAccess.Read,
FileShare.None);
Header h = Header.ReadHeader(bf);
long n = h.DataSize;
int  naxes  = h.GetIntValue("NAXIS");
```

```
int  lastAxis = h.GetIntValue("NAXIS"+naxes);
HeaderCard hnew= new HeaderCard("NAXIS", naxes
- 1,"this is header card with naxes");
 h.AddCard(hnew);
 float[] line = new float[h.DataSize];
 for (int i = 0; i < lastAxis; i += 1)
 {
         Console.Out.WriteLine("read");
         bf.Read(line);
         Process a line.
 }
```

Here's what we did: First we opened the file directly as a BufferedFile rather than using the Fits class. We read only the first header. Then things got devious. We modified the header so that it described something with one less dimension than originally. E.g., if the header originally had described a 2-D real array it now described only one line of that array. Using this modified header we got a data object so that we could get an array of the appropriate dimensions and type to read the data. Then we read the image data explicitly using the methods of BufferedFile.

Is there any reason for this subterfuge? Probably not but is illustrates a little of the inner workings of the FITS classes.

Open a Fits file using specified FileAccess Mode:

We can open a Fits file by specifying user defined mode of opening the file like:

```
f = new Fits("bt2.fits", FileAccess.Read);
f = new Fits("bt2.fits", FileAccess.Write);
```

Read a compressed file:

If the file is Gzip compressed and ends in '.gz', just use the file name and this will be handled automatically. If not you may want to do something like:

```
File fl = new File("filename");
Fits f = new Fits(fl, true);
```

The second argument indicates that the File is compressed.

Note that deferred reads and re-writes are not supported for compressed files.

Read a URL:

Just enter the full URL for HTTP protocol URLs. You can also use the URL constructors.

```
Fits f = new
Fits("http://xyz.edu/somedata.fits");
```

or

```
URL u = new
URL("http://xyz.edu/somedcompressedFitsData.ff"
);
Fits f = new Fits(u, true);
```

Read an ASCII table:

At its simplest reading an ASCII table almost as easy as an image. One big difference though is that tables can never be the primary data for a FITS file.

The data for an ASCII table is returned as a set of arrays, one for each column in the table.

```
Fits f = new
Fits("fileWithAsciiTable",FileAccess.Read);
BasicHDU[] hdus = f.Read();

float[] realCol = new float[50];
for (int i = 0; i < realCol.Length; i+= 1)
{
     realCol[i] = 10000F * (i) * (i) * (i) + 1;
}
int[] intCol = Array.ConvertAll<float,
int>(realCol, Convert.ToInt32);
long[] longCol….
```

```
    … these arrays have the same dimensions …
    Object[] cols =new Object[] { realCol, intCol,
    longCol, doubleCol, strCol };
```

Read pieces of an ASCII table:

Read pieces of ASCII table by row:

```
AsciiTableHDU hdu = (AsciiTableHDU)f.GetHDU(1);
AsciiTable data = (AsciiTable)hdu.GetData();
for (int i = 0; i < data.NRows; i += 1)
{
     Object[] row = (Object[])data.GetRow(i);
}
```

One vital thing to note: When referring to FITS rows and columns, the FITS library uses the 0-based indices for these, not 1-based FITS indices. I.e., the column that has 'TFORM1' is column 0. This is confusing. Alas, the alternatives seem just as bad.

You can also just get a single element:

```
        Object val = data.GetElement(29,30);
```

And of course one can get a column:

```
Object col = data.GetColumn(j);
```

Reading rows and elements does not require reading the entire extension. AsciiTables support deferred reads.

Large ASCII tables can take a long time to read but fortunately they are relatively rare.

Writing an ASCII table:

If you have a set of simple String, int, long, float and double arrays it's easy to write an ASCII table.

```
    Object[] baseObj = new Object[] {arr1, arr2,
arr3, …};
      Fits f = new Fits();
```

```
      FitsFactory.SetUseAsciiTables(true);
    f.AddHDU(Fits.MakeHDU(baseObj));
    BufferedFile bf = new
BufferedFile("outputfile", "rw");
    bf.write(bf);
      bf.flush();
      bf.close();
```

Note that we told the system that we wanted to have ASCII tables. That's the current default, but it never hurts to make sure! Also note that we added only the ASCII table to the Fits object. The Fits object inserted a dummy primary HDU for us automatically.

Modifying an ASCII table:

There are methods corresponding to the getRow, getColumn and getElement routines to allow the user to modify the contents of existing rows and columns. A user can also build a table column by column.

```
AsciiTable d = new AsciiTable();
d.addColumn(anArray);
d.addColumn(anotherArray);
…
Header h = AsciiTableHDU.manufactureHeader(d);
AsciiTableHDU newAscii = AsciiTableHDU(h,d);
```

Now that we're ready to add it to a FITS object and write it to output.

We could also have used a SetColumn (or SetRow or SetElement) method to modify existing data. E.g.,

```
Fits f = new Fits("aFileName");
AciiTableHDU hdu = (AsciiTableHDU)f.GetHDU(1);
  AsciiTable data = (AsciiTable)hdu.GetData();
  float[] f1 = (float[])data.GetColumn(0);
float[] f2 = (float[])f1.Clone();
data.SetColumn(0, f2);
  f1 = new float[] { 3.14159f };
data.SetElement(3, 0, f1);
```

```
Object[] row = new Object[5];
data.SetRow(5, row);
data.SetElement(4, 2, new long[] { 54321 });
```

ASCII table null values:

ASCII tables allow the user to specify that a value is null whenever the ASCII pattern in the file matches a specified null pattern. To check if an element is null you can use the IsNull call. Also, the GetElement() and GetRow() methods will return a null for null values. However numeric columns return a single primitive array, so they just have a 0 for nulls. You need to use the IsNull methods to check if you find a null, if you get data using GetColumn or GetKernel.

To set a field to null, just use the SetNull method. Make sure that TNULL is defined for the appropriate column.

Precision in ASCII tables:

The ASCII tables use the ByteParser and ByteFormatter routines to provide efficient conversions between binary and ASCII representations of numbers. These conversions may differ in the lowest bit from those provided by the standard system utilities – which are much slower.

When writing real values, ASCII tables use only the length field in determining what to write. They ignore the number of decimals specified for real numbers. This will likely be changed in the future but should normally mean more accurate representations and fewer occasions where numbers overflow their fields. The tables are likely to be less neatly aligned however.

When creating header information for ASCII tables, the headers use a default value for the TFORM for each type. I10, I20, E16.0 and E24.0 for integer, long, float and double numbers respectively. Strings are checked for the longest string and are formatted for that to fit. Users can override these values as desired by specifying the TFORM values explicitly. If rows are added to ASCII tables the existing field lengths are preserved. If String cannot fit into the space provided it is truncated. If no sensible representation of a number can fit into the space provided a set of asterisks is inserted.

Short Integer Types in ASCII tables:

There is currently no support for byte, short or char arrays in reading or writing ASCII tables.

Reading Binary Tables:

Reading a binary table is very similar to an ASCII table, but the kernel data type is very different. The kernel is a new class called a ColumnTable. When I first attempted to read binary tables I found that they were extremely slow. There were too distinct reason why reading binary tables is extremely slow: First, binary tables often are comprised of small heterogeneous elements, e.g., an int followed by a double[2] followed by a short. Thus the ArrayDataIO methods cannot read the data efficiently. These are designed to be efficient when the data is comprised of large arrays.

The second problem is that if we store the data naively, with each element in the array as separate object, we may need to create millions of objects. Just creating these objects can cause a noticeable pause in a FITS-reading program.

We don't have this problem with ASCII tables because it's quite easy to represent each column in the table as an array, so there are only as many objects as there are columns. In ASCII tables each element must be a scalar, but for binary tables each element can be an array of any dimensionality.

The ColumnTable addresses these two issues. It stores data in a fashion similar to the FITS AsciiTable class, each column is represented by a single one-dimensional array. However each column also has associated with it an integer array describing the dimensionality of the array. The ColumnTable can read and write complete rows in a single function call. The ColumnTable is not a FITS class. It can be used to read and write other binary tabular data. When users actually retrieve data from a ColumnTable they can choose to either get it as the one-d array or to 'curl' elements into their nominal dimensionality.

Let's take a look:

```
Fits f = new Fits("binaryTableFile");
```

```
BinaryTableHDU h = (BinaryTableHDU)
f.GetHDU(1);

Object[] row23 = h.GetRow(23);
Object   col3  = h.GetColumn(3);
Object   col3f = h.GetFlattenedColumn(3);
Object   elem  = h.GetElement(10,20);
ColumnTable t  = (ColumnTable) h.Kernel();
```

Note that the GetRow method returned an array of objects, one for each column in the table. If an element is itself an array this will be appropriately curled.

The GetColumn call returned a curled array. E.g., if the third column is a 3x2 array and there are 500 rows in the table, the col3 might be int[500][3][2]. Unlike getColumn in the AsciiTable methods, we can't modify the FITS file by looking at this array. This is a copy of the internal data, not a reference to it.

The GetFlattenedColumn is a little more efficient. For the same example it would return an int[3000] which is how the data is stored internally. Indeed, the getFlattenedArray returns you a pointer to the actual FITS array data. You can modify the FITS file by modifying this array – we're getting this array directly from the ColumnTable kernel.

The GetElement returns a single element, curled if the element is a multidimensional array. Even if the column contains scalars, the returned object will be a length=1 array. The scalar types like Integer are never returned by the FITS classes. However a scalar String's will be returned as a String, not a String[1].

The GetRow and GetElement methods allow deferred reading, but the GetColumn methods read in the entire table. The getter for Kernel also ensures that the all data are in memory.

Reading a binary table by row:

Both binary and ASCII HDU types extend the type TableHDU. Binary tables support the same basic get, set and add operations as ASCII tables. Users can also access binary table data at a lower level. Here's one idiom.

```
BufferedDataStream s = new
BufferedDataStream(…);
Fits f = new Fits(s);
f.GetHDU(0);                                  // Read
primary HDU.
Header hdr     = Header.ReadHeader(s);  // Read
table header
BinaryTable bt = new BinaryTable(hdr);
Object[] row   = bt.GetSampleRow();
int nrow       = hdr.GetIntValue("NAXIS2");

for (int i=0; i< nrow; i += 1) {
      s.ReadArray(row);
      … Process a line.
}
```

Each element of `row` is an array of the appropriate type and dimensionality for the binary table elements.

Reading variable length columns:

While not strictly part of standard FITS variable length columns are extensively used by some organizations (mine included). These allow a table to have a column that varies in length row by row. Variable length columns are only supported as one-dimensional arrays.

Variable-length columns are handled automatically by the FITS reader. Users may note that one or more elements might have zero-length. The isVariableColumn method allows the user to see if a column is specified as variable – but does not actually check if it varies.

Note that the data for variable length columns is not stored in the ColumnArray that is returned as the kernel of the binary table. There is a FitsHeap object associated with any table with variable length columns. The BinaryTableHDU add, get, and set methods handle variable length arrays automatically. If the user wants to do devious tricks, e.g., have many rows

in a table point to the same binary table data, then methods in ColumnTable and FitsHeap will need to be called directly.

When variable length columns are read, no aliasing information is retained. A distinct array is created for every row in the table when the variable length column is read, even if the data was originally
aliased.

Writing binary tables:

A binary table HDU can be created from two distinct types of data arrays of Objects and existing ColumnTables that the user has created. An Object[][] arrays is used to specify each element of the table.
An Object[] array is used to specify a binary table as an array of columns in much the same fashion as ASCII tables. Note that some caution is needed to ensure that the appropriate constructors are called. If a binary table is to be composed of simple arrays, then the user may want to disable the creation of ASCII tables – or create the BinaryTable HDU directly by calls to the appropriate functions and constructors in BinaryTableHDU.

```
          FitsFactory.UseAsciiTables = false;

              Fits f = new Fits();
          Object[] data = new Object[]{bytes, bits,
          bools, shorts,  ints,
            floats, doubles, longs, strings};
              f.AddHDU(Fits.MakeHDU(data));

              BinaryTableHDU bhdu =
(BinaryTableHDU)f.GetHDU(1);
              bhdu.SetColumnName(0, "bytes", null);
              bhdu.SetColumnName(1, "bits", "bits
later on");
              bhdu.SetColumnName(6, "doubles", null);
              bhdu.SetColumnName(5, "floats", "4 x 4
array");

          BufferedFile bf = new
          BufferedFile("bt1.fits",
```

```
FileAccess.ReadWrite,
FileShare.ReadWrite);
    f.Write(bf);
    bf.Flush();
    bf.Close();
```

Binary table internals

Users can treat ASCII and binary tables using much the same interface, but internally they are very different. There are several different data structures that are used internally within binary tables, and for the most efficient access to binary tables a user needs to understand a bit of this.

A binary table column is an n-dimensional array, where the first dimension is the number of rows in the table. The additional dimensions are what would be specified in the TDIM field for that column.

If a column is a two-dimensional primitive array the second dimension may be variable. This is represented as a varying length column in the binary table HDU. E.g.,
short byte[][] = {{1},{1,2},{1,2,3}}
might be used in a varying length column with three rows.

A flattened column is a one-dimensional array where we have unrolled the other dimensions. E.g., suppose we have TFORM8 = '80E' and TDIM8='(2,2,2,10)' in a table with 100 rows. The column would be represented as a float[100][10][2][2][2]. When converted to a flattened column it would be a float[8000]. Flattened columns are used to minimize the creation of array objects. Variable length columns cannot be flattened.

A model row is an exact image of how a row in the binary table is stored. In the above example, it would consist of an Object[] where the eighth element was a float[10][2][2][2]. Note that model rows represent how the data is stored in the FITS file. Since Strings and booleans are converted into byte arrays on output to the binary table, this is how they appear in a model row. Also, variable length columns are represented not by their actual data, but by a two element int descriptor array.

The object which is normally used to read and write binary table data is the ColumnTable. This basically is a structure of flattened columns transformed to the data type used in the binary table. I.e.,
String and boolean data has been converted into bytes, and variable length columns have been converted into descriptor arrays.

Variable length data is stored in a FitsHeap. Whenever data is written to a variable length column the heap is extended. Whenever a user reads variable length columns, the descriptor information is used to extract data from the heap.

Unknown data types.

The FITS library supports the reading and writing of data types that are not fully understood by the FITS library. Any FITS extension that follows the rules for indicating the size of the HDU can be read. Any data structure that can be written using the writeArray method of ArrayDataOutput can be encapsulated in a FITS extension. This includes multi-dimensional (possibly non-rectangular) arrays of primitives, Strings or Objects, where any Object must itself be one of these types. Using Object arrays an arbitrarily complex structure can be supported.

Data is read into a byte array or is written using an XTENSION='UNKNOWN'. Note that one cannot trivially save and restore data in a FITS file. If a user has some structure it can be written to a FITS file using the Unknown type, but to read it back the user will need to be able to recreate the data structure without help from the FITS file.

Users may find these classes useful if there are non-standard extensions in their FITS data. The FITS reader can successfully bypass these unknown types and read later extensions. Also see the section on trimming the installation.

blocks in the header).

Objects and I/O Streams

In the examples above the user may be confused since we sometimes have calls of the form

```
hdu.Write(stream);  and others
stream.Write(data);
```

When do we send the object to the stream, and when do we send the stream to the object? Basically if the object is a FITS object, it will take a stream as an argument for input or output. If the object is an array—comprised of only Objects, Strings, and primitive types, then this object can be sent to the DataArrayIO implementors for reading or writing.