FITS Classes – Modeling a Standard

Allen Farris
Space Telescope Science Institute

Note
This document assumes a general familiarity with the structure of a FITS file. The best summary of current FITS standards is given in [1], which contains extensive discussion and further references within the astronomical literature to various papers regarding FITS.

1 Introduction

A FITS file, in general, is a sequence of header-data-units. The file as a whole and the structure of each header-data-unit is subject to strict formatting rules. The header portion of a header-data-unit consists of ASCII text that describes the structure and content of the data portion, which immediately follows. This header portion consists of an arbitrary number of eighty byte sequences of keyword name, data value and comment in the form "keyword = value / comment". Keywords are divided into standard and user-defined; standard keywords are also either mandatory or optional. Mandatory keywords identify the type of header-data-unit, the data type, and the data structure and size. Optional keywords include scaling factors, units, and keywords to aid in the physical interpretation of the data. Supported data structures include multi-dimensional arrays, random groups, ASCII tables, binary tables, and the image extension.

This document gives an overview of a set of C++ classes designed to aid in the construction of programs to read and write FITS files. They also aim to strictly adhere to the FITS standards. On input, errors or failures to adhere to FITS standards are flagged but reasonable attempts are made to correct these deficiencies and continue processing. On output, every attempt to enforce FITS standards is made; the classes will not knowingly write a file that does not adhere to FITS standards.

This set of classes assumes a sequential processing model in processing the FITS file, whether on input or output. This is consistent with the FITS standards, which define a linear sequence of header-data-units followed by an optional sequence of "special" records. In other words, no assumptions are made which require the processed file to be on random access media. The header-data-units are processed in the linear order in which they occur within the file. The primary purpose of this set of classes is to allow the user to write programs that import data into or export data from some internal format.

2 Class Structure

The FITS classes fall naturally into two groups, the classes that directly model the header-data-units themselves and the classes that form the underlying infrastructure that manipulates the FITS records, parses the header records, enforces the FITS formatting rules, and relates the header-data-units to these records.

The structure of the header-data-units is depicted in figure one. The class HeaderDataUnit contains what is common to all header-data-units, including the collection of keywords. This keyword collection is in the form of a linked list of objects of the Keyword class, which contains the name, value and any comment. On input, this linked list is formed from the FITS header; on output, it is used to form the FITS header. There are an extensive number of functions for manipulating this linked list of keywords. Classes derived from the HeaderDataUnit are specific types of FITS header-data-units: PrimaryArray, PrimaryGroup, ImageExtension, ExtensionHeaderDataUnit, BinaryTableExtension, and AsciiTableExtension. Each one of these has a rich assortment of functions for accessing and manipulating data of specific types.

The classes that form the underlying infrastructure are depicted in figure two. The highest level class, and the most important to the user, is FitsIO. The HeaderDataUnit class, from figure one, uses the FitsIO class for its I/O. FitsIO is the class that contains the knowledge of how to translate FITS fixed length records into keyword lists and blocks of data. It uses a class, FitsKeyCardTranslator, to translate between Keyword lists and the fixed FITS "card" format. The FITS parser is contained within the FitsKeyCardTranslator. In addition to providing this translation service, FITSIO keeps track of vital I/O information, such as the current type of header-data-unit, the current record type, size of remaining data, etc. The classes FitsInput and FitsOutput, derived from FitsIO, are the classes that users will instantiate to identify the FITS file being read or written. FitsInput also has a function allowing a reader to skip an entire header-data-unit.

FitsInput and FitsOutput use classes that implement fixed-length blocked I/O, the BlockIO, BlockInput, and BlockOutput classes. Specific physical devices supported by these classes are derived from BlockInput and BlockOutput. They include disk I/O, standard I/O, and 9-track tape I/O. By isolating physical devices in this manner, other types of media, such as optical disks or helical scan tapes, are easy to add to the list of supported devices.

One final class must be mentioned in the overview, the FITS class. This class consists of a series of static functions and enumerations. Many of the static functions are utility functions used internally in the implementation of the member functions of the FITS classes. They are placed in a single class to encapsulate them and to avoid adding many names to the global namespace. More important, from the user's perspective, is the enumerations. They form the basic vocabulary of a FITS application. The names in the enumerations may be used by prepending FITS:: to the names, e. g., to refer to the FITS NAXIS keyword FITS::NAXIS should be used. They are used in the following manner. Suppose we have a FITS input file, as in the following declaration.

```
FitsInput fin("myfile.fit",FITS::Disk);
```

Then, a test for the primary FITS array might be the following.

```
if (fin.rectype() == FITS::HDURecord &&
    fin.hdutype() == FITS::PrimaryArrayHDU)
```

The enumerations in the FITS class are listed below.

Table of Enumerations in the FITS Class

```
ValueType // Basic FITS Data Types for keywords and data
    NOVALUE   LOGICAL   BIT       CHAR      BYTE
    SHORT     LONG      FLOAT     DOUBLE    COMPLEX
    ICOMPLEX  DCOMPLEX  VADESC    STRING

ReservedName // FITS Reserved Names
    USER_DEF  AUTHOR    BITPIX    BLANK     BLOCKED   BSCALE
    BUNIT     BZERO     CDELT     COMMENT   CROTA     CRPIX
    CRVAL     CTYPE     DATAMAX   DATAMIN   DATE      DATE_OBS
    END       EPOCH     EQUINOX   EXTEND    EXTLEVEL  EXTNAME
    EXTVER    GCOUNT    GROUPS    HISTORY   INSTRUME  NAXIS
    OBJECT    OBSERVER  ORIGIN    PCOUNT    PSCAL     PTYPE
    PZERO     REFERENC  SIMPLE    SPACES    TBCOL     TDIM
    TDISP     TELESCOP  TFIELDS   TFORM     THEAP     TNULL
    TSCAL     TTYPE     TUNIT     TZERO     XTENSION  ERRWORD

FitsRecType // Types of FITS Records
    InitialState          BadBeginningRecord  HDURecord
    UnrecognizableRecord  SpecialRecord       EndOfFile

FitsDevice // Supported FITS Physical Devices
    Disk  Std  Tape9

HDUType // Types of FITS Header-Data Units
    NotAHDU         PrimaryArrayHDU    PrimaryGroupHDU
    AsciiTableHDU   BinaryTableHDU     ImageExtensionHDU
    UnknownExtensionHDU

FitsArrayOption // Options on FITS array manipulations
    NoOpt   CtoF   FtoC
```

3 Access to Data

The technique for accessing data will be illustrated using the FITS primary array, as implemented by the PrimaryArray class. This class fully supports arrays of arbitrary dimensionality, up to the FITS limit of 999 dimensions. For a PrimaryArray, dims() gives the number of dimensions and dim(i) gives the value of the i-th dimension. FITS multi-dimensional arrays are stored in FORTRAN order, not in C order. Options on the store, copy, and move functions exist to convert from one order to the other, if that is necessary. The PrimaryArray class is a template class, with the TYPE parameter being the type of data in the FITS array. The public portion of the PrimaryArray class is shown below.

```
template <class TYPE>
class PrimaryArray : public HeaderDataUnit {
  public:
    PrimaryArray(FitsInput &, ostream & = cout);
    PrimaryArray(FitsKeywordList &, ostream & = cout);
    virtual ~PrimaryArray();

    double bscale() const;
    double bzero() const;
    char *bunit() const;
    Bool isablank() const;
    Int blank() const;
    char *ctype(int n) const;
    double crpix(int n) const;
    double crota(int n) const;
    double crval(int n) const;
    double cdelt(int n) const;
    double datamax() const;
    double datamin() const;
    uInt nelements() const;

    double operator () (int, int, int ...) const; // return physical data
    double operator () (int, int) const;
    double operator () (int) const;

    TYPE & data(int, int, int ...); // access raw data
    TYPE & data(int, int);
    TYPE & data(int);

    int store(const TYPE *source, FITS::FitsArrayOption = FITS::NoOpt);
    void copy(double *target, FITS::FitsArrayOption = FITS::NoOpt) const;
    void copy(float *target, FITS::FitsArrayOption = FITS::NoOpt) const;
    void move(TYPE *target, FITS::FitsArrayOption = FITS::NoOpt) const;

    virtual int read(); // read entire array into memory
```

```
    virtual int read(int); // read next N elements into memory

    virtual int write(FitsOutput &); // write current data
    virtual int set_next(int); // prepare to write next N elements
};
```

The overloaded operator functions '()' all return physical data, i. e., data to which bscale() and bzero() have been applied, via the formula:

```
    physical_data[i] = bscale() * raw_data[i] + bzero().
```

The various 'data()' functions allow one to access and set the raw data itself.

The 'store()', 'move()' and 'copy()' functions allow bulk data transfer between the internal FITS array and an external data storage area. The external storage must have already been allocated and it is assumed that the entire data array is in memory. 'Store()' transfers raw data at 'source' into the FITS array; an allowable option is CtoF, which specifies to convert the array from C-order to Fortran-order. 'Move()' is the opposite of 'store()'. 'Move()' transfers raw data from the FITS array to 'target'; an allowable option is FtoC, which specifies to convert the array from Fortran-order to C-order. 'Copy()' is similar to 'move()' except that what is copied is physical data and not raw data; the physical data can be either double or float.

The 'read()' and 'write()' functions control reading and writing data from the external FITS I/O medium into the FITS array. Appropriate conversions are made between FITS and local data representations. One can read the entire array into memory, or one can only read portions of the array. In the latter case, one must specify that the next N elements are to be read or written. Note that the number of elements must be specified, NOT the number of bytes. If one reads portions of the array, as opposed to the entire array, only that portion is in memory at a given time. One can still access the elements of the array via the '()' and 'data()' functions, as if the entire array was in memory; obviously care must be taken in this case to access only those portions that are actually in memory.

It is important to understand the proper sequence of operations with respect to I/O and data access. For input, the 'read()' functions allocate an internal buffer of the appropriate size, if not already allocated, as well as reading and converting data; a 'read()' function must be performed prior to accessing the data, i. e. before executing any '()', 'data()', 'copy()', or 'move()' function. For output, the 'store()' function similarly allocates an internal buffer before transferring data, and must be executed prior to any data access or 'write()' function.

Writing portions of an array at a time, rather than the entire array, is a special case. The 'set_next()' function is provided for this purpose. It declares the intention to write out the next N elements and must be executed prior to any 'data()' function. It allocates a buffer of appropriate size, if not already allocated. Again, via the 'data()' functions, one accesses the

array as if the entire array were in memory. The 'write()' function always writes the number of current elements in the internal buffer. The sequence of operations for each portion of the array written would be: 1) 'set_next(N)', 2) fill the array using 'data(N)' or other 'data()' functions, and 3) 'write(fout)'. The 'set_next()' function must NOT be used with 'read()' or 'store()' functions; unpredictable results will occur. The following example illustrates the output cases.

Suppose we have an image array with 512 rows and 1024 columns stored in C-order. The C declaration would be:

```
int source[1024][512];
```

To write out the entire array:

```
FitsOutput fout; // some properly constructed FitsOutput
PrimaryArray<int> pa; // some properly constructed PrimaryArray
pa.store(source,FITS::CtoF);
pa.write(fout);
```

Suppose we wanted to write out the two-dimensional image array a column at a time, rather than write out the entire array. For FITS, dim(0) is 512, dim(1) is 1024. The following code fragment writes one column at a time in the proper FITS Fortran-order.

```
for (i = 0; i < dim(1); ++i) {
    pa.set_next(dim(0));
    for (j = 0; j < dim(0); ++j)
        data(j,i) = source[i][j];
    pa.write(fout);
}
```

4 An Example Program

The following program creates a FITS file containing a primary image and an image extension. A brief commentary follows the program source code.

```
# include <aips/fits.h>
# include <aips/fitsio.h>
# include <aips/hdu.h>
# include <iostream.h>
# include <stdlib.h>
```

```
int main() {
    cout << "Create a primary array and an image extension\n";
    FitsOutput fout("test.dat",FITS::Disk);
    if (fout.err() == FitsIO::IOERR) {
        cout << "Could not open FITS output.\n";
        exit(0);
    }

    // We will create an array with 3 rows and 6 columns,
    // as one would normally do in C.
    const int row = 3;
    const int col = 6;
    long data[col][row];
    // And, we will populate it with some kind of data
    int i, j;
    for (i = 0; i < col; ++i)
      for(j = 0; j < row; ++j)
        data[i][j] = j * 10 + i;

    FitsKeywordList st; // Create the initial keyword list
    st.mk(FITS::SIMPLE,True,"Standard FITS format");
    st.mk(FITS::BITPIX,32,"Integer data");
    st.mk(FITS::NAXIS,2,"This is a primary array");
    st.mk(1,FITS::NAXIS,3);
    st.mk(2,FITS::NAXIS,6);
    st.mk(FITS::EXTEND,True,"Extension exists");
    st.spaces();
    st.comment("This is a test.");
    st.spaces();
    st.end();

    PrimaryArray<long> hdu1(st); // Create and write the initial HDU
    if (hdu1.err())
        exit(0);
    cout << "Initial HDU constructed\n";
    cout << hdu1; // Display the keyword list
    hdu1.write_hdr(fout);
    hdu1.store(&data[0][0],FITS::CtoF);
    hdu1.write(fout);

    FitsKeywordList kw; // Create a keyword list for an image extension
    kw.mk(FITS::XTENSION,"IMAGE   ","Image extension");
    kw.mk(FITS::BITPIX,32,"Integer data");
    kw.mk(FITS::NAXIS,2,"This is an image");
    kw.mk(1,FITS::NAXIS,3);
    kw.mk(2,FITS::NAXIS,6);
    kw.mk(FITS::PCOUNT,0);
```

```
    kw.mk(FITS::GCOUNT,1);
    kw.spaces();
    kw.comment("This is test 2.");
    kw.spaces();
    kw.end();

    ImageExtension<long> ie(kw);
    if (ie.err())
        exit(0);
    cout << "ImageExtension constructed\n";
    cout << ie; // Display the keyword list
    ie.write_hdr(fout);

    ie.set_next(row * col);        // setup to write the whole array
    for (i = 0; i < row; ++i)
      for(j = 0; j < col; ++j)
        ie.data(i,j) = i * 10 + j; // assign the data

    ie.write(fout);                // write the data

    return 0;
}
```

The 'FitsOutput' statement creates a FITS disk file whose name is 'test.dat'. The following section allocates a 3x6 array of integers and assigns some arbitrary data to the array. The section beginning 'FitsKeywordList st' declares a keyword list, st. The function, st.mk(), creates a keyword entry and appends it to the end of the list. So, the following sequence of st functions, down to st.end(), creates a linked list of keywords sufficient to define a primary array with two dimensions, with 3 rows and 6 columns of integer data.

The statement, 'PrimaryArray¡long¿ hdu1(st);', creates a primary array header-data-unit, hdu1, using the st keyword list. The statement, 'hdu1.write_hdr(fout);' translates the keyword list to a FITS header and writes it to the previously created FitsOutput file, fout. The 'hdu1.store' function copies the C data array into the FITS buffer, while converting it from C-order to FORTRAN-order. Finally, the 'hdu1.write(fout)' function writes the FITS data to disk, thus completing the writing of the primary array.

The next sequence of code creates another keyword list that defines an image extension. The statement 'ImageExtension¡long¿ ie(kw);' creates the image extension and 'ie.write_hdr(fout)' writes the FITS header, as before. The following code uses the 'set_next' function to allocate a buffer area and the 'data' function to assign data to it. This sequence of operations is functionally equivalent to the 'store' function used in the previous section. Finally, the 'ie.write' function writes the data to disk.

References

[1] FITS 1993, *"Definition of the Flexible Image Transport System (FITS)"*, Standard, NOST 100-1.0, NASA/Science Office of Standards and Technology, Code 633.2, NASA Goddard Space Flight Center, Greenbelt, Maryland