HEASP Guide

Version 1.0

Keith A. Arnaud

HEASARC
Code 662
Goddard Space Flight Center
Greenbelt, MD 20771
USA

Mar 2012

Contents

Chapter 1

Introduction

HEASP is a C/C++/Python library to manipulate files associated with high energy astrophysics spectroscopic analysis. Currently this handles PHA, RMF, ARF and xspec table model files. The eventual plan is to be able to provide at least the functionality available in current ftools but from a single library.

The main code is written in C++ with classes for each file type. These classes and associated methods have been run through SWIG to produce Python code. C wrappers are provided for many of the C++ methods allowing simple use from C programs.

These C wrappers are compatible with an earlier C only version of HEASP with two exceptions: a) the include file required is now Cheasp.h instead of heasp.h; b) all routines which used to require a FITS file pointer now just require the filename. One consequence of b) is that files no longer need to be opened and closed by the calling program.

There are separate chapters describing the Python, C++, and C interfaces although users of Python should consult the C++ chapter for description of the classes.

Highlights of HEASP are :

- Read and write spectra, responses, arfs and table model files.
- Rebin spectra using grouping arrays.
- Compress responses by removing all elements below some value. Rebin responses in either energy or channel space based on a grouping array.
- Use a response to generate random channel numbers for a photon of a given energy.
- Sum both rmfs and arfs and multiply rmfs by arfs.
- Construct type II PHA or ARF from sets of spectra or arfs. Also extract individual spectra or arfs from type II files.

Chapter 2

Python module

2.1 Getting started

The standard HEAsoft build makes a Python module based on the HEASP C++ classes described in the next chapter. The standard HEAsoft initialization script adds the location of the HEASP Python module to PYTHONPATH. To load the module use

```
UNIX> python
>>>from heasp import *
```

If this produces name conflicts then loading the module by

```
UNIX> python
>>>import heasp
```

requires heasp. in front of all commands.

To set up an HEASP object first give a simple command such as

```
>>>sp = pha()
```

for a spectrum. This can then be read in and information about it displayed by

```
>>>sp.read("file1.pha")
>>>sp.disp()
```

Individual components of the object can be used by adding the component name to the object name without "()". For instance:

```
>>>first = sp.FirstChannel
```

will place the first channel number in the variable first. The components of each class are given in the C++ chapter.

Components which are arrays can be used only through their individual element numbers. For instance:

```
>>>counts = []
>>>for i in xrange(sp.NumberChannels()):
...   counts.append(sp.Pha[i])
```

will place the contents of the PHA column in the list called counts. Going the other way, the current contents of the PHA column can be replaced by those in the counts list by:

```
>>>for i in xrange(sp.NumberChannels()):
...   sp.Pha[i] = counts[i]
```

2.2 HEASP features not (yet) supported in Python

At present the HEASP Python module does not support binary operations defined in the C++ classes such as the addition of two responses by resp = resp1 + resp2. What are supported are the unary equivalents so responses can be added by resp1 += resp2.

2.3 Spectrum example

The following Python example reads a type II PHA file, rebins the channels in each spectrum by a factor of 2 and writes out the result.

```
# read the spectrum
spectra = phaII()
spectra.read("testin.pha")

# loop round the spectra in the file
# rebinning by a factor of 2 then placing
# in the output spectra
output = phaII()
Nspectra = spectra.NumberSpectra()
groupInfo = grouping()

for i in xrange(Nspectra):
  spectrum = spectra.get(i)
  groupInfo.load(2,spectrum.NumberChannels())
  status = spectrum.rebinChannels(groupInfo)
  output.push(spectrum)
```

```
# write out the spectrum copying any extra keywords and extensions
# from testin.pha
output.write("testout.pha", "testin.pha")
```

2.4 Response example

The following Python example reads RMF and ARF files, removes all elements smaller than 10^{-6} from the RMF, multiplies the compressed RMF and the ARF, and writes out the result.

```
# read RMF and ARF
inputRMF = rmf()
inputRMF.read("testin.rmf")
inputARF = arf()
inputARF.read("testin.arf")

# compress the RMF
inputRMF.compress(1.0e-6)

# if the RMF and ARF are compatible then multiply them and write
# the result adding extra keywords and extensions from testin.rmf.
if inputRMF.checkCompatibility(inputARF):
  inputRMF *= inputARF
  inputRMF.write("testout.rsp", "testin.rmf")
```

2.5 Table model example

The following Python example sets up a table model grid with two parameters and one additional parameter (for instance abundance). The parameters, energies and fluxes are given arbitrary values, in practice these could be read from text files.

```
test = table()

# set table descriptors and the energy array
test.ModelName = "Test"
test.ModelUnits = " "
test.isRedshift = True
test.isAdditive = True
test.isError = False

# set up the energies. note that the size is one greater
# than that of the array for the model fluxes
```

```
for i in xrange(100): test.Energies.append(0.1+i*0.1)

test.NumIntParams = 2
test.NumAddParams = 1

# define first parameter and give it 11 values ranging from
# 0.0 to 2.0 in steps of 0.2.

testpar = tableParameter()
testpar.Name = "param1"
testpar.InterpolationMethod = 0
testpar.InitialValue = 1.0
testpar.Delta = 0.1
testpar.Minimum = 0.0
testpar.Bottom = 0.0
testpar.Top = 2.0
testpar.Maximum = 2.0

for i in xrange(11): testpar.TabulatedValues.append(0.2*i)

# and push it onto the vector of parameters
test.pushParameter(testpar)

# define the second parameter and give it 5 values ranging from
# 4.6 to 5.4 in steps of 0.2.

testpar.clear()
testpar.Name = "param2"
testpar.InterpolationMethod = 0
testpar.InitialValue = 5.0
testpar.Delta = 0.1
testpar.Minimum = 4.6
testpar.Bottom = 4.6
testpar.Top = 5.4
testpar.Maximum = 5.4

for i in xrange(11): testpar.TabulatedValues.append(4.6+0.2*i)

# and push it onto the vector of parameters
test.pushParameter(testpar);

# define an additional parameter (usually the elemental abundance)
# does not require tabulated values.

testpar.clear()
```

```
testpar.Name = "addparam"
testpar.InterpolationMethod = 0
testpar.InitialValue = 0.0
testpar.Delta = 0.1
testpar.Minimum = 0.0
testpar.Bottom = 0.0
testpar.Top = 5.0
testpar.Maximum = 5.0

# and push it onto the vector of parameters
test.pushParameter(testpar)

# now set up the spectra. these are arbitrarily calculated, in a real program
# this step would read a file or call a routine.

addflux = []
testspec = tableSpectrum()

for i1 in xrange(11):
  for i2 in xrange(5):
    testspec.clear()
    testspec.ParameterValues.append(0.2*i1)
    testspec.ParameterValues.append(4.6+0.2*i2)
    addflux = []
    for j in xrange(99):
      testspec.Flux.append(0.2*i1+10*(4.6+0.2*i2)+j*0.1)
      addflux.append((i1+1)*(i2+1)+j*0.2)
    testspec.pushaddFlux(addflux)
    test.pushSpectrum(testspec)

# now write out the table.
test.write("test.mod");
```

Chapter 3

C++ classes and methods

3.1 Spectra

3.1.1 Introduction and example

Spectrum files can be manipulated using the pha and phaII classes. The latter is simply a vector of pha classes and is useful for handling type II PHA files. The grouping class and utility routines are also useful for some tasks. As an example the code below reads in a type II PHA file, bins up all the spectra by a factor of 2, then writes out the result. Note that the data types Integer and Real are defined in heasp.h.

```
#include "grouping.h"
#include "phaII.h"

using namespace std;

int main(int argc, char* argv[])
{
  string infile("testin.pha");
  string outfile("testout.pha");

  phaII inputSpectra;

  Integer Status(0);


  // read in all the spectra

  Status = inputSpectra.read(infile, 1);
```

```
  Integer Nspectra = inputSpectra.NumberSpectra();

  // loop round the spectra

  for (size_t i=0; i<(size_t)Nspectra; i++) {

    // set up the grouping object to rebin by a factor of 2

    grouping groupInfo;
    groupInfo.load(2, inputSpectra.phas[i].NumberChannels());

    // rebin this spectrum

    Status = inputSpectra.phas[i].rebinChannels(groupInfo);

  }

  // write the new spectra out copying extra keywords and extensions from
  // the input file

  Status = inputSpectra.write(outfile, infile);

  exit(0);
}
```

3.1.2 pha class

```
class pha{
 public:

  Integer FirstChannel;     // First legal channel number

  vector<Real> Pha;         // PHA data
  vector<Real> StatError;   // Statistical error
  vector<Real> SysError;    // Statistical error

  vector<Integer> Channel;  // Channel number
  vector<Integer> Quality;  // Data quality
  vector<Integer> Group;    // Data grouping

  vector<Real> AreaScaling; // Area scaling factor
  vector<Real> BackScaling; // Background scaling factor
```

```
Real Exposure;            // Exposure time
Real CorrectionScaling;   // Correction file scale factor

Integer DetChans;         // Total legal number of channels
bool Poisserr;            // If true, errors are Poisson
string Datatype;          // "COUNT" for count data and "RATE" for count/sec
string PHAVersion;        // PHA extension format version

string Spectrumtype;      // "TOTAL", "NET", or "BKG"

string ResponseFile;      // Response filename
string AncillaryFile;     // Ancillary filename
string BackgroundFile;    // Background filename
string CorrectionFile;    // Correction filename

string ChannelType;       // Value of CHANTYPE keyword
string Telescope;
string Instrument;
string Detector;
string Filter;
string Datamode;

vector<string> XSPECFilter; // Filter keywords
```

3.1.3 pha class public methods

- ```
  Integer read(string filename)

  Integer read(string filename, Integer PHAnumber)

  Integer read(string filename, Integer PHAnumber,
      Integer SpectrumNumber)
  ```

  Read file into object. If PHAnumber is given then look for the SPECTRUM extension with EXTVER=PHAnumber. The third option is to read the pha from the SpectrumNumber row of a type II file.

- `pha& operator= (const pha&)`

  Deep copy.

- `Integer NumberChannels()`

  Return the size of vector¡Real¿s.

- `string disp()`

  Display information about the spectrum - return as a string.
  ```

- `void clear()`

  Clear information from the spectrum

- `string check()`

  Check completeness and consistency of information in spectrum, if there is a problem then return diagnostic in string.

- ```
  Integer write(string filename)

  Integer write(string filename, string copyfilename)

  Integer write(string filename, string copyfilename,
        Integer HDUnumber)
  ```

  Write spectrum as type I file. If copyfilename is given then copy from it other HDUs and other keywords in the SPECTRUM extension. If HDUnumber is specified then use the SPECTRUM extension with EXTVER=HDUnumber in copyfilename. Note that if the output filename exists and already has a SPECTRUM extension then these methods will write an additional SPECTRUM extension.

- `void setGrouping(grouping&, Integer&)`

  Set the pha grouping array from a grouping object.

- `void rebinChannels(grouping&, Integer&)`

  Rebin spectrum channels based on a grouping object.

3.1.4 Other pha routines

- ```
  Integer PHAtype(string filename, Integer PHAnumber)

  Integer PHAtype(string filename, Integer PHAnumber,
        Integer& Status)
  ```

  Return the type of a SPECTRUM extension.

- ```
  bool IsPHAcounts(string filename, Integer PHAnumber)

  bool IsPHAcounts(string filename, Integer PHAnumber,
        Integer& Status)
  ```

  Return true if the COUNTS column exists and is integer.

- ```
  Integer NumberofSpectra(string filename,
        Integer PHAnumber)

  Integer NumberofSpectra(string filename,
        Integer PHAnumber, Integer& Status)
  ```

  Return the number of spectra in a type II SPECTRUM extension.

3.1.5 phaII class

```
class phaII{
 public:

  vector<pha> phas;           // vector of pha objects
```

3.1.6 phaII class public methods

- `Integer read(string filename)`

  `Integer read(string filename, Integer PHAnumber)`

  `Integer read(string filename, Integer PHAnumber,`
  `vector<Integer> SpectrumNumber)`

  Read a PHA type II file into an object. If PHAnumber is given then read from the SPECTRUM extension with EXTVER=PHAnumber. If the SpectrumNumber array is given then read those rows in the extension otherwise read all spectra.

- `phaII& operator= (const phaII&)`

  Deep copy.

- `pha get(Integer number)`

  Get pha object (counts from zero).

- `pha push(pha sp)`

  Push pha object into phaII object

- `Integer NumberSpectra()`

  Return the number of spectra in the object.

- `string disp()`

  Display information about the spectra - return as a string.

- `void clear()`

  Clear information about the spectra

- `string check()`

  Check completeness and consistency of information in spectrum, if there is a problem then return diagnostic in string.

- `Integer write(string filename)`

```
Integer write(string filename, string copyfilename)

Integer write(string filename, string copyfilename,
      Integer HDUnumber)
```

Write spectra as type II file. If copyfilename is given then copy from it other HDUs and other keywords in the SPECTRUM extension. If HDUnumber is specified then use the SPECTRUM extension with EXTVER=HDUnumber in copyfilename. Note that if the output filename exists and already has a SPECTRUM extension then these methods will write an additional SPECTRUM extension.

3.2 Responses

3.2.1 Introduction and example

Response files come in two varieties: RMFs and ARFs. The former contain the response matrices describing the probability of a photon of a given energy being registered in a given channel of the spectrum. The latter describes the effictive area versus energy. The rmf class is used for manipulating RMFs and the arf and arfII classes for manipulating ARFs. The arfII class is an analog of the phaII class and is useful for the case where an ARF file contains many individual effective area curves. The rmft class handles the transposed response matrix and is of limited use at present.

The example code below shows a program to read in an RMF file, to compress the matrix to remove any element below 1.0e-6, to multiply the result by an ARF, and write a new RMF file.

```
#include "rmf.h"
#ifndef HAVE_arf
#include "arf.h"
#endif

using namespace std;

int main(int argc, char* argv[])
{
  string rmffile("testin.rmf");
  string arffile("testin.arf");
  string outfile("testout.rmf");

  rmf inputRMF, outputRMF;
  arf inputARF;

  Integer Status(0);
```

```
  // read in the RMF and the ARF

  Status = inputRMF.read(rmffile);
  Status = inputARF.read(arffile);

  // remove elements from the RMF with values < 1.0e-6

  Real threshold(1.0e-6);
  inputRMF.compress(threshold);

  // multiply the compressed RMF and the ARF to make an output RMF

  if ( inputRMF.checkCompatibility(inputARF) ) {

    outputRMF = inputRMF * inputARF;

    // and write out the result copying any extra HDUs and keywords from
    // the input RMF

    Status = outputRMF.write(outfile, rmffile);

  }

  exit(0);
}
```

3.2.2 rmf class

```
class rmf{
 public:

  Integer FirstChannel;                // First channel number

  vector<Integer> NumberGroups;        // Number of response groups for this
                                       //  energy bin
  vector<Integer> FirstGroup;          // First response group for this energy
                                       // bin (counts from 0)

  vector<Integer> FirstChannelGroup;   // First channel number in this group
  vector<Integer> NumberChannelsGroup; // Number of channels in this group
  vector<Integer> FirstElement;        // First response element for this group
                                       // (counts from 0)
```

```
vector<Integer> OrderGroup;          // The grating order of this group

vector<Real> LowEnergy;              // Start energy of bin
vector<Real> HighEnergy;             // End energy of bin

vector<Real> Matrix;                 // Matrix elements

vector<Real> ChannelLowEnergy;       // Start energy of channel
vector<Real> ChannelHighEnergy;      // End energy of channel

Real AreaScaling;                    // Value of EFFAREA keyword
Real ResponseThreshold;              // Minimum value in response

string EnergyUnits;                  // Energy units used
string RMFUnits;                     // Units for RMF values

string ChannelType;                  // Value of CHANTYPE keyword
string RMFVersion;                   // MATRIX extension format version
string EBDVersion;                   // EBOUNDS extension format version
string Telescope;
string Instrument;
string Detector;
string Filter;
string RMFType;                      // HDUCLAS3 keyword in MATRIX extension
string RMFExtensionName;             // EXTNAME keyword in MATRIX extension
string EBDExtensionName;             // EXTNAME keyword in EBOUNDS extension
```

3.2.3 rmf class public methods

- `Integer read(string filename)`

  `Integer read(string filename, Integer RMFnumber)`

  Read the RMF file into an object. If RMFnumber is given read from the MATRIX (or SPECRESP MATRIX) and EBOUNDS extensions with EXTVER=RMFnumber. If there is only one EBOUNDS extension then that will be used.

- `Integer readMatrix(string filename)`

  `Integer readMatrix(string filename, Integer RMFnumber)`

  Read the MATRIX (or SPECRESP MATRIX) extension from an RMF file into an object. If RMFnumber is given read from the MATRIX (or SPECRESP MATRIX) extension with EXTVER=RMFnumber.

- `Integer readChannelBounds(string filename)`

  `Integer readChannelBounds(string filename, Integer RMFnumber)`

  Read the EBOUNDS extension from an RMF file into an object. If RMFnumber is given read from the EBOUNDS extension with EXTVER=RMFnumber.

- `void update()`

  Update the FirstGroup and FirstElement arrays from NumberGroups and NumberChannels-Group, respectively.

- `void initialize(const arf&)`

  Initialize from an arf object. Copies members in common between arfs and rmfs

- `rmf& operator= (const rmf&)`

  Deep copy.

- `Integer NumberChannels()`

  Return the number of spectrum channels.

- `Integer NumberEnergyBins()`

  Return the number of response energies.

- `Integer NumberTotalGroups()`

  Return the number of response groups.

- `Integer NumberTotalElements()`

  Return the number of response elements.

- `Real ElementValue(Integer Channel, Integer EnergyBin)`

  Return the value for a particular channel and energy.

- `vector<Real> RowValues(Integer EnergyBin)`

  Return the response array for a particular energy

- ```
  vector<Integer> RandomChannels(const Real energy,
        const Integer NumberPhotons)
  ```

  Use the response matrix to generate random channel numbers for a photon of given energy.

- `string disp()`

  Display information about the response. - return as a string.

- `void clear()`

  Clear information from the response.
  ```

- `string check()`

  Check completeness and consistency of information in the rmf, if there is a problem then return diagnostic in string.

- `void normalize()`

  Normalize the rmf so it sums to 1.0 for each energy bin.

- `void compress(const Real threshold)`

  Compress the rmf to remove all elements below the threshold value.

- `Integer rebinChannels(grouping&)`

  Rebin in channel space using the specified grouping object.

- `Integer rebinEnergies(grouping&)`

  Rebin in energy space using the specified grouping object.

- ```
  Integer write(string filename)

  Integer write(string filename, string copyfilename)

  Integer write(string filename, string copyfilename,
        Integer HDUnumber)
  ```

  Write response to a RMF file. If copyfilename is given then copy from it other HDUs and other keywords in the MATRIX and EBOUNDS extensions. If HDUnumber is specified then use the MATRIX and EBOUNDS extensions with EXTVER=HDUnumber in copyfilename. Note that if the output filename exists and already has MATRIX and EBOUNDS extensions then these methods will write additional extensions.

- ```
  Integer writeMatrix(string filename)

  Integer writeMatrix(string filename, string copyfilename)

  Integer writeMatrix(string filename, string copyfilename,
        Integer HDUnumber)
  ```

  Write the MATRIX extension to a RMF file. If copyfilename is given then copy from it other HDUs and other keywords in the MATRIX extension. If HDUnumber is specified then use the MATRIX extension with EXTVER=HDUnumber in copyfilename. Note that if the output filename exists and already has a MATRIX extension then these methods will write an additional extension.

- ```
  Integer writeChannelBounds(string filename)

  Integer writeChannelBounds(string filename, string copyfilename)
  ```
  ```

```
Integer writeChannelBounds(string filename, string copyfilename,
      Integer HDUnumber)
```

Write the EBOUNDS extension to a RMF file. If copyfilename is given then copy from it other HDUs and other keywords in the EBOUNDS extension. If HDUnumber is specified then use the EBOUNDS extension with EXTVER=HDUnumber in copyfilename. Note that if the output filename exists and already has a EBOUNDS extension then these methods will write an additional extension.

- `rmf& operator*=(const arf&)`

  Multiply current rmf by an arf.

- `rmf& operator+=(const rmf&)`

  Add another rmf to the current rmf.

- `bool checkCompatibility(const rmf&)`

  Check compatibility with another rmf.

- `bool checkCompatibility(const arf&)`

  Check compatibility with an arf.

3.2.4 Other rmf routines

- `rmf operator* (const rmf&, const arf&)`

  `rmf operator* (const arf&, const rmf&)`

  Multiply an rmf by an arf.

- `rmf operator+ (const rmf&, const rmf&)`

  Add two rmfs.

- ```
  void compressLine(const vector<Real> Response,
        const Integer FirstChannel, const Real threshold,
        Integer& NumberGroups, vector<Integer>& oFirstChGrp,
        vector<Integer>& oNumberChsGrp, vector<Real>& oMatrix)
  ```

  Useful routine used in constructing an rmf from individual response arrays for each energy. Takes in an array and adds on to the rmf.
  ```

3.2.5 rmft class

```
class rmft{
 public:

  Integer FirstChannel;                // First channel number

  vector<Integer> NumberGroups;        // Number of response groups for this
                                       // channel bin
  vector<Integer> FirstGroup;          // First response group for this channel
                                       // bin (counts from 0)

  vector<Integer> FirstEnergyGroup;    // First energy bin in this group
  vector<Integer> NumberEnergiesGroup; // Number of energy bins in this group
  vector<Integer> FirstElement;        // First response element for this group
                                       // (counts from 0)
  vector<Integer> OrderGroup;          // The grating order of this group

  vector<Real> LowEnergy;              // Start energy of bin
  vector<Real> HighEnergy;             // End energy of bin

  vector<Real> Matrix;                 // Matrix elements

  vector<Real> ChannelLowEnergy;       // Start energy of channel
  vector<Real> ChannelHighEnergy;      // End energy of channel

  Real AreaScaling;                    // Value of EFFAREA keyword
  Real ResponseThreshold;              // Minimum value in response

  string EnergyUnits;                  // Energy units
  string RMFUnits;                     // RMF units

  string ChannelType;                  // Value of CHANTYPE keyword
  string RMFVersion;                   // MATRIX extension format version
  string EBDVersion;                   // EBOUNDS extension format version
  string Telescope;
  string Instrument;
  string Detector;
  string Filter;
  string RMFType;                      // HDUCLAS3 keyword in MATRIX extension
  string RMFExtensionName;             // EXTNAME keyword in MATRIX extension
  string EBDExtensionName;             // EXTNAME keyword in EBOUNDS extension
```

3.2.6 rmft class public methods

- `void load(rmf&)`

  Load object from a standard rmf object.

- `void update`

  Update the FirstGroup and FirstElement arrays from NumberGroups and NumberEnergies-Group, respectively.

- `rmft& operator= (const rmft&)`

  Deep copy.

- `Integer NumberChannels()`

  Number of spectrum channels

- `Integer NumberEnergyBins()`

  Number of response energies

- `Integer NumberTotalGroups()`

  Total number of response groups

- `Integer NumberTotalElements()`

  Total number of response elements

- `Real ElementValue(Integer Channel, Integer EnergyBin)`

  Return the value for a particular channel and energy.

- `vector<Real> RowValues(Integer Channel)`

  Return the array for a particular channel.

- `string disp()`

  Display information about the object. - return as a string.

- `void clear()`

  Clear information from the object.

3.2.7 arf class

```
class arf{
 public:

  vector<Real> LowEnergy;   // Start energy of bin
  vector<Real> HighEnergy;  // End energy of bin
```

```
vector<Real> EffArea;     // Effective areas

string EnergyUnits;       // Units for energies
string arfUnits;          // Units for effective areas

string Version;           // SPECRESP extension format version
string Telescope;
string Instrument;
string Detector;
string Filter;
string ExtensionName;     // EXTNAME keyword in SPECRESP extension
```

3.2.8 arf class public methods

- `Integer read(string filename)`

  `Integer read(string filename, Integer ARFnumber)`

  ```
  Integer read(string filename, Integer ARFnumber,
      Integer RowNumber)
  ```

  Read file into an object. If ARFnumber is given read from the SPECRESP extension with EXTVER=ARFnumber. The third option is to read an arf from the RowNumber row of a type II file.

- `arf& operator= (const arf&)`

  Deep copy.

- `Integer NumberEnergyBins()`

  Return size of vector¡Real¿s.

- `string disp()`

  Display information about the arf. - return as a string.

- `void clear()`

  Clear information from the arf.

- `string check()`

  Check completeness and consistency of information in the arf, if there is a problem then return diagnostic in string.

- `Integer write(string filename)`

  `Integer write(string filename, string copyfilename)`

```
Integer write(string filename, string copyfilename,
      Integer HDUnumber)
```

Write arf as type I file. If copyfilename is given then copy from it other HDUs and other keywords in the SPECRESP extension. If HDUnumber is specified then use the SPECRESP extension with EXTVER=HDUnumber in copyfilename. Note that if the output filename exists and already has a SPECRESP extension then these methods will write an additional SPECRESP extension.

- `arf& operator+=(const arf&)`

  Add another arf.

- `bool checkCompatibility(const arf&)`

  Check compatibility with another arf.

3.2.9 Other arf routines

- `arf operator+ (const arf&, const arf&)`

  Add two arfs.

- `Integer NumberofARFs(string filename, Integer HDUumber)`

  ```
  Integer NumberofARFs(string filename, Integer HDUumber,
       Integer& Status)
  ```

  Return the number of ARFS in the type II SPECRESP extension.

3.2.10 arfII class

```
class arfII{
 public:

  vector<arf> arfs;           // vector of arf objects
```

3.2.11 arfII class public methods

- `Integer read(string filename)`

  ```
  Integer read(string filename, Integer ARFnumber)
  ```

```
Integer read(string filename, Integer ARFnumber,
      vector<Integer> RowNumber)
```

Read an ARF type II file into an object. If ARFnumber is given then read from the SPECRESP extension with EXTVER=ARFnumber. If the RowNumber array is given then read those in the extension otherwise read all the arfs.

- `arfII& operator= (const arfII&)`

  Deep copy.

- `arf get(Integer number)`

  Get arf object (counts from zero).

- `void push(arf ea)`

  Push arf object into arfII object

- `Integer NumberARFs()`

  Return the number of ARFs in the object.

- `string disp()`

  Display information about the ARFs. - return as a string.

- `void clear()`

  Clear information from the ARFs.

- `string check()`

  Check completeness and consistency of information in the arfs, if there is a problem then return diagnostic in string.

- `Integer write(string filename)`

  ```
  Integer write(string filename, string copyfilename)
  ```

  ```
  Integer write(string filename, string copyfilename,
        Integer HDUnumber)
  ```

  Write ARFs as type II file. If copyfilename is given then copy from it other HDUs and other keywords in the SPECRESP extension. If HDUnumber is specified then use the SPECRESP extension with EXTVER=HDUnumber in copyfilename. Note that if the output filename exists and already has a SPECRESP extension then these methods will write an additional SPECRESP extension.

3.3 Table Models

3.3.1 Introduction and example

The table model file is used in xspec to provide grids of model calculations on which to interpolate when fitting a model to data. The table class can be used to create these files. The example code below sets up a grid with two parameters.

```
#include "table.h"

using namespace std;

int main(int argc, char* argv[])
{

  table test;

  // set table descriptors and the energy array

  test.ModelName = "Test";
  test.ModelUnits = " ";
  test.isRedshift = true;
  test.isAdditive = true;
  test.isError = false;

  test.Energies.resize(100);
  for (size_t i=0; i<100; i++) test.Energies[i] = 0.1+i*0.1;

  test.NumIntParams = 2;
  test.NumAddParams = 1;


  // define first parameter and give it 11 values ranging from
  // 0.0 to 2.0 in steps of 0.2.

  tableParameter testpar;

  testpar.Name = "param1";
  testpar.InterpolationMethod = 0;
  testpar.InitialValue = 1.0;
  testpar.Delta = 0.1;
  testpar.Minimum = 0.0;
  testpar.Bottom = 0.0;
  testpar.Top = 2.0;
  testpar.Maximum = 2.0;
```

```
testpar.TabulatedValues.resize(11);
for (size_t i=0; i<11; i++) testpar.TabulatedValues[i] = 0.2*i;

// and push it onto the vector of parameters

test.Parameters.push_back(testpar);

// define the second parameter and give it 5 values ranging from
// 4.6 to 5.4 in steps of 0.2.


testpar.Name = "param2";
testpar.InterpolationMethod = 0;
testpar.InitialValue = 5.0;
testpar.Delta = 0.1;
testpar.Minimum = 4.6;
testpar.Bottom = 4.6;
testpar.Top = 5.4;
testpar.Maximum = 5.4;

testpar.TabulatedValues.resize(5);
for (size_t i=0; i<5; i++) testpar.TabulatedValues[i] = 4.6+0.2*i;

// and push it onto the vector of parameters

test.Parameters.push_back(testpar);

// define an additional parameter (usually the elemental abundance)
// does not require tabulated values.

testpar.Name = "addparam";
testpar.InterpolationMethod = 0;
testpar.InitialValue = 0.0;
testpar.Delta = 0.1;
testpar.Minimum = 0.0;
testpar.Bottom = 0.0;
testpar.Top = 5.0;
testpar.Maximum = 5.0;
testpar.TabulatedValues.resize(0);

// and push it onto the vector of parameters

test.Parameters.push_back(testpar);
```

```
  // now set up the spectra. these are arbitrarily calculated, in a real program
  // this step would read a file or call a routine.

  tableSpectrum testspec;

  testspec.Flux.resize(99);
  testspec.ParameterValues.resize(2);

  vector<Real> addFlux(99);

  for (size_t i1=0; i1<11; i1++) {
    for (size_t i2=0; i2<5; i2++) {
      testspec.ParameterValues[0] = 0.2*i1;
      testspec.ParameterValues[1] = 4.6+0.2*i2;
      for (size_t j=0; j<99; j++) {
testspec.Flux[j] = testspec.ParameterValues[0]+10*testspec.ParameterValues[1];
addFlux[j] = 1.0*(i1+1)*(i2+1);
      }
      testspec.addFlux.push_back(addFlux);
      test.Spectra.push_back(testspec);
      testspec.addFlux.clear();
    }
  }

  // now write out the table.

  test.write("test.mod");

  exit(0);
}
```

3.3.2 table classes

The class for parameters:

```
class tableParameter{
 public:

  string Name;                  // Parameter name
  int InterpolationMethod;      // 0==linear, 1==log
  Real InitialValue;            // Initial value for fit
  Real Delta;                   // Delta for fit
  Real Minimum;                 // Hard lower-limit
                                // (should correspond to first tabulated value)
```

```
  Real Bottom;                  // Soft lower-limit
  Real Top;                     // Soft upper-limit
  Real Maximum;                 // Hard upper-limit
                                // (should correspond to last tabulated value)
  vector<Real> TabulatedValues; // Tabulated parameter values
```

and its public methods:

- `string disp()`
  Display information about the table parameter - return as a string.
- `void clear()`
  Clear contents of the table parameter

The class for model spectra:

```
class tableSpectrum{
 public:

  vector<Real> Flux;               // Model flux
  vector<Real> ParameterValues;    // Parameter values for this spectrum
  vector<vector<Real>> addFlux;    // Model fluxes for any additional
                                   // parameters
```

and its public methods:

- `void pushaddFlux(vector<Real>)`
  Push an additional parameter spectrum
- `vector<Real> getaddFlux(Integer Number)`
  Get an additional parameter spectrum
- `string disp()`
  Display information about the table spectrum - return as a string.
- `void clear()`
  Clear contents of the table spectrum

Finally, the class for the complete table:

```
class table{
 public:
```

```
vector <tableParameter> Parameters; // Parameter information
vector <tableSpectrum> Spectra;     // Tabulated model spectra
string ModelName;                   // Name to use in xspec
string ModelUnits;                  // Units (not used at present)
int NumIntParams;                   // Dimension of interpolation grid
int NumAddParams;                   // Number of additional parameters
bool isError;                       // If true then model errors included
bool isRedshift;                    // If true include redshift
bool isAdditive;                    // If true model is additive
vector<Real> Energies;              // Energy bins on which model is calculated
                                    // The size should be one larger than that
                                    // of the spectrum array
```

and its public methods:

- `void pushParameter(tableParameter paramObject)`

  Push a table parameter object

- `void pushSpectrum(tableSpectrum spectrumObject)`

  Push a table spectrum object

- `tableParameter getParameter(Integer Number)`

  Get a table parameter object

- `tableSpectrum getSpectrum(Integer Number)`

  Get a table spectrum object

- `string disp()`

  Display information about the table - return as a string.

- `void clear()`

  Clear contents of the table

- `string check()`

  Check completeness and consistency of information in table, if there is a problem then return diagnostic in string.

- `write(string filename)`

  Write to a FITS file

3.4 Grouping

3.4.1 grouping class

```
class grouping{
 public:

  vector<Integer> flag;     // Grouping flag: 1=start of bin,
                            //                0=continuation of bin
```

3.4.2 grouping class public methods

- `grouping(vector<Integer> flaginput)`

  Constructor from an integer array of flag values.

- `string disp()`

  Display grouping information. - return as a string.

- `void clear()`

  Clear grouping information.

- `Integer read(string filename, const Integer Number,`
  `const Integer First)`

  Read from an ascii file of grouping factors. Each line of the file should have three numbers, the start bin, end bin, and grouping factor. The input bin numbers start at First and there are Number in total.

- `void load(const Integer BinFactor, const Integer Number)`

  Set the grouping flags for Number bins with a binning factor of BinFactor.

- `Integer load(const vector<Integer>& StartBin, const vector<Integer>& EndBin,`
  `const vector<Integer>& BinFactor, const Integer Number,`
  `const Integer First)`

  Set grouping flags from an array of binning information in the StartBin, EndBin and BinFactor arrays. The input bin numbers start at First and there are Number in total.

- `bool newBin(const Integer Bin)`

  Return whether Bin is the start of a group.

- `Integer size()`

  Return number of bins in grouping object.

3.4.3 Other grouping routines

- `template <class T> void GroupBin(const vector<T>& inArray,`
  `const Integer mode, const grouping& GroupInfo, vector<T>& outArray)`

  `template <class T> void GroupBin(const valarray<T>& inArray,`
  `const Integer mode, const grouping& GroupInfo, valarray<T>& outArray)`

  This routine applies GroupInfo to the input inArray to create the output outArray. The behavior is determined by mode which can take five values: SumMode which adds the contents of all bins in a group; SumQuadMode which adds the bins in quadrature; MeanMode which returns the arithmetic mean of the all bins in a group; FirstEltMode which returns the value of the first bin in each group; LastEltMode which returns the value of the last bin in each group.

- `Integer readBinFactors(string filename, vector<Integer>& StartBin,`
  `vector<Integer>& EndBin, vector<Integer>& BinFactor)`

  Read a file containing grouping information and place into the arrays StartBin, EndBin and BinFactor. Each line of the file should have three numbers, the start bin, end bin, and grouping factor.

3.5 Utility routines

- `void SPreadColUnits(ExtHDU&, string, string&)`

  Read the units associated with a column.

- `void SPwriteColUnits(Table&, string, string)`

  Write the units associated with a column.

- `string SPstringTform(const vector<string>& Data)`

  Returns the tform string for the longest string in the input vector.

- `Integer SPcopyHDUs(string infile, string outfile)`

  Copy from infile to outfile all HDUs which are not manipulated by this library.

- `Integer SPcopyKeys(string infile, string outfile, string HDUname,`
  `Integer HDUnumber)`

  Copy non-critical keywords from infile to outfile for HDUname extension with EXTVER HDUnumber.

Chapter 4

C interface

4.1 PHA files

4.1.1 PHA structure

```
struct PHA {

  long NumberChannels;                /* Number of spectrum channels */
  long FirstChannel;                  /* First channel number */

  float* Pha;/*NumberChannels*/       /* PHA data */
  float* StatError;/*NumberChannels*/ /* Statistical error */
  float* SysError;/*NumberChannels*/  /* Statistical error */

  int*   Quality;/*NumberChannels*/   /* Data quality */
  int*   Grouping; /*NumberChannels*/ /* Data grouping */
  int*   Channel; /*NumberChannels*/  /* Channel number */

  float* AreaScaling;/*NumberChannels*//* Area scaling factor */
  float* BackScaling;/*NumberChannels*//* Background scaling factor */

  float Exposure;                     /* Exposure time */
  float CorrectionScaling;            /* Correction file scale factor */
  int   DetChans;                     /* Content of DETCHANS keyword */

  int Poisserr;                       /* If true, errors are Poisson */
  char Datatype[FLEN_KEYWORD];        /* "COUNT" for count data and */
                                      /* "RATE" for count/sec */
  char Spectrumtype[FLEN_KEYWORD];    /* "TOTAL", "NET", or "BKG" */
```

```
  char ResponseFile[FLEN_FILENAME];    /* Response filename */
  char AncillaryFile[FLEN_FILENAME];   /* Ancillary filename */
  char BackgroundFile[FLEN_FILENAME];  /* Background filename */
  char CorrectionFile[FLEN_FILENAME];  /* Correction filename */

  char ChannelType[FLEN_KEYWORD];      /* Value of CHANTYPE keyword */
  char PHAVersion[FLEN_KEYWORD];       /* PHA extension format version */
  char Telescope[FLEN_KEYWORD];
  char Instrument[FLEN_KEYWORD];
  char Detector[FLEN_KEYWORD];
  char Filter[FLEN_KEYWORD];
  char Datamode[FLEN_KEYWORD];

  char *XSPECFilter[100];              /* Filter keywords */
};
```

4.1.2 PHA routines

- `int ReadPHAtypeI(char *filename, long PHAnumber, struct PHA *phastruct)`

  Read the type I SPECTRUM extension from a FITS file - if there are multiple SPECTRUM extensions then read the one with EXTVER=PHAnumber.

- `int ReadPHAtypeII(char *filename, long PHAnumber, long NumberSpectra, long *SpectrumNumber, struct PHA **phastructs)`

  Read the type II SPECTRUM extension from a FITS file - if there are multiple SPECTRUM extensions then read the one with EXTVER=PHAnumber. Within the SPECTRUM extension reads the spectra listed in the SpectrumNumber vector.

- `int WritePHAtypeI(char *filename, struct PHA *phastruct)`

  Write the spectrum to a type I SPECTRUM extension in a FITS file.

- `int WritePHAtypeII(char *filename, long NumberSpectra, struct PHA **phastructs)`

  Write the multiple spectra to a type II SPECTRUM extension in a FITS file.

- `int ReturnPHAtype(char *filename, long PHAnumber)`

  Return the type of the SPECTRUM extension with EXTVER=PHAnumber.

- `void DisplayPHAtypeI(struct PHA *phastruct)`

  Write information about the spectrum to stdout.

- `void DisplayPHAtypeII(long NumberSpectra, struct PHA **phastructs)`

  Write information about multiple spectra to stdout.

- `int RebinPHA(struct PHA *phastruct, struct BinFactors *bin)`

  Rebin spectrum.

- `int CheckPHAcounts(char *filename, long PHAnumber)`

  Return 0 if COUNTS column exists and is integer or COUNTS column does not exist.

- `long ReturnNumberofSpectra(char *filename, long PHAnumber)`

  Return the number of spectra in the type II SPECTRUM extension which has EXTVER equal to PHAnumber.

4.2 RMF files

4.2.1 RMF structure

```
struct RMF {

  long NumberChannels;                          /*Number of spectrum channels*/
  long NumberEnergyBins;                        /*Number of response energies*/
  long NumberTotalGroups;                       /*Total number of resp groups*/
  long NumberTotalElements;                     /*Total number of resp elts*/
  long FirstChannel;                            /*First channel number*/
  long isOrder;                                 /*If true grating order*/
                                                /*information included*/

  long* NumberGroups;/*NumberEnergyBins*/       /*Number of resp groups for*/
                                                /*this energy bin*/
  long* FirstGroup;/*NumberEnergyBins*/         /*First resp group for this*/
                                                /*energy bin (counts from 0)*/

  long* FirstChannelGroup;/*NumberTotalGroups*/ /*First channel number in*/
                                                /*this group*/
  long* NumberChannelGroups;/*NumberTotalGroups*//*Num of channels in this grp*/
  long* FirstElement;/*NumberTotalGroups*/      /*First resp elt for this grp*/
                                                /*(counts from 0)*/
  long* OrderGroup;/*NumberTotalGroups*/        /*Grating order of this grp*/

  float* LowEnergy;/*NumberEnergyBins*/         /*Start energy of bin*/
  float* HighEnergy;/*NumberEnergyBins*/        /*End energy of bin*/

  float* Matrix;/*NumberTotalElements*/         /*Matrix elements*/

  float* ChannelLowEnergy;/*NumberChannels*/    /*Start energy of channel*/
  float* ChannelHighEnergy;/*NumberChannels*/   /*End energy of channel*/
```

```
  float AreaScaling;                          /*Value of EFFAREA keyword*/
  float ResponseThreshold;                    /*Minimum value in response*/

  char EnergyUnits[FLEN_KEYWORD];             /*Units for energies*/
  char RMFUnits[FLEN_KEYWORD];                /*Units for RMF*/

  char ChannelType[FLEN_KEYWORD];             /*Value of CHANTYPE keyword*/
  char RMFVersion[FLEN_KEYWORD];              /*MATRIX format version*/
  char EBDVersion[FLEN_KEYWORD];              /*EBOUNDS format version*/
  char Telescope[FLEN_KEYWORD];
  char Instrument[FLEN_KEYWORD];
  char Detector[FLEN_KEYWORD];
  char Filter[FLEN_KEYWORD];
  char RMFType[FLEN_KEYWORD];                 /*HDUCLAS3 keyword in MATRIX*/
  char RMFExtensionName[FLEN_VALUE];          /*EXTNAME keyword in MATRIX*/
  char EBDExtensionName[FLEN_VALUE];          /*EXTNAME keyword in EBOUNDS*/

};

struct RMFchan {

  long NumberChannels;                       /*Number of spectrum channels*/
  long NumberEnergyBins;                     /*Number of response energies*/
  long NumberTotalGroups;                    /*Total number of resp groups*/
  long NumberTotalElements;                  /*Total number of resp elts*/
  long FirstChannel;                         /*First channel number*/
  long isOrder;                              /*If true grating order*/
                                             /*information included*/

  long* NumberGroups;/*NumberChannels*/      /*Number of resp groups for*/
                                             /*this channel bin*/
  long* FirstGroup;/*NumberChannels*/        /*First resp group for this*/
                                             /*channel bin (counts from 0)*/

  long* FirstEnergyGroup;/*NumberTotalGroups*/  /*First energy bin in this grp*/
  long* NumberEnergyGroups;/*NumberTotalGroups*//*Number of energy bins in*/
                                             /*this group */
  long* FirstElement;/*NumberTotalGroups*/   /*First resp elt for this grp*/
                                             /*(counts from 0)*/
  long* OrderGroup;/*NumberTotalGroups*/     /*Grating order of this group*/

  float* LowEnergy;/*NumberEnergyBins*/      /*Start energy of bin*/
  float* HighEnergy;/*NumberEnergyBins*/     /*End energy of bin*/
```

```
  float* Matrix;/*NumberTotalElements*/          /*Matrix elements*/

  float* ChannelLowEnergy;/*NumberChannels*/     /*Start energy of channel*/
  float* ChannelHighEnergy;/*NumberChannels*/    /*End energy of channel*/

  float AreaScaling;                             /*Value of EFFAREA keyword*/
  float ResponseThreshold;                       /*Minimum value in response*/

  char EnergyUnits[FLEN_KEYWORD];                /*Units for energies*/
  char RMFUnits[FLEN_KEYWORD];                   /*Units for RMF*/

  char ChannelType[FLEN_KEYWORD];                /*Value of CHANTYPE keyword*/
  char RMFVersion[FLEN_KEYWORD];                 /*MATRIX format version*/
  char EBDVersion[FLEN_KEYWORD];                 /*EBOUNDS format version*/
  char Telescope[FLEN_KEYWORD];
  char Instrument[FLEN_KEYWORD];
  char Detector[FLEN_KEYWORD];
  char Filter[FLEN_KEYWORD];
  char RMFType[FLEN_KEYWORD];                    /*HDUCLAS3 keyword in MATRIX*/
  char RMFExtensionName[FLEN_VALUE];             /*EXTNAME keyword in MATRIX*/
  char EBDExtensionName[FLEN_VALUE];             /*EXTNAME keyword in EBOUNDS*/

};
```

4.2.2 RMF routines

- `int ReadRMFMatrix(char *filename, long RMFnumber, struct RMF *rmf)`

  Read the RMF matrix from a FITS file - if there are multiple RMF extensions then read the one with EXTVER=RMFnumber.

- `int WriteRMFMatrix(char *filename, struct RMF *rmf)`

  Write the RMF matrix to a FITS file.

- `int ReadRMFEbounds(char *filename, long EBDnumber, struct RMF *rmf).`

  Read the RMF ebounds from a FITS file - if there are multiple EBOUNDS extensions then read the one with EXTVER=EBDnumber.

- `int WriteRMFEbounds(char *filename, struct RMF *rmf)`

  Write the RMF ebounds to a FITS file.

- `void DisplayRMF(struct RMF *rmf)`

  Write information about RMF to stdout.

- `void ReturnChannel(struct RMF *rmf, float energy, int NumberPhotons, long *channel)`

  Return the channel for a photon of the given input energy - draws random numbers to return NumberPhotons entries in the channel array.

- `void NormalizeRMF(struct RMF *rmf)`

  Normalize the response to unity in each energy.

- `void CompressRMF(struct RMF *rmf, float threshold)`

  Compress the response to remove all elements below the threshold value.

- `int RebinRMFChannel(struct RMF *rmf, struct BinFactors *bins)`

  Rebin the RMF in channel space.

- `int RebinRMFEnergy(struct RMF *rmf, struct BinFactors *bins)`

  Rebin the RMF in energy space.

- `void TransposeRMF(struct RMF *rmf, struct RMFchan *rmfchan)`

  Transpose the matrix.

- `float ReturnRMFElement(struct RMF *rmf, long channel, long energybin)`

  Return a single value from the matrix.

- `float ReturnRMFchanElement(struct RMFchan *rmfchan, long channel, long energybin)`

  Return a single value from the transposed matrix.

- `int AddRMF(struct RMF *rmf1, struct RMF *rmf2)`

  Add rmf2 onto rmf1.

4.3 ARF files

4.3.1 ARF structure

```
struct ARF {

  long NumberEnergyBins;                  /* Number of response energies */

  float* LowEnergy; /*NumberEnergyBins*/  /* Start energy of bin */
  float* HighEnergy; /*NumberEnergyBins*/ /* End energy of bin */

  float* EffArea;    /*NumberEnergyBins*/ /* Effective areas */

  char EnergyUnits[FLEN_KEYWORD];         /* Units for energies */
```

```
  char arfUnits[FLEN_KEYWORD];            /* Units for effective areas */

  char ARFVersion[FLEN_KEYWORD];          /* SPECRESP extension format version */
  char Telescope[FLEN_KEYWORD];
  char Instrument[FLEN_KEYWORD];
  char Detector[FLEN_KEYWORD];
  char Filter[FLEN_KEYWORD];
  char ARFExtensionName[FLEN_VALUE];      /* EXTNAME keyword in SPECRESP */

};
```

4.3.2 ARF routines

- `int ReadARF(char *filename, long ARFnumber, struct ARF *arf)`

  Read the effective areas from a FITS file - if there are multiple SPECRESP extensions then read the one with EXTVER=ARFFnumber.

- `int WriteARF(char *filename, struct ARF *arf)`

  Write the ARF to a FITS file.

- `void DisplayARF(struct ARF *arf)`

  Write information about ARF to stdout.

- `int AddARF(struct ARF *arf1, struct ARF *arf2)`

  Add arf2 onto arf1.

- `long MergeARFRMF(struct ARF *arf, struct RMF *rmf)`

  Multiply the ARF into the RMF.

4.4 Binning and utility

4.4.1 BinFactors structure

```
struct BinFactors {

  long NumberBinFactors;

  long *StartBin;
  long *EndBin;
  long *Binning;

};
```

4.4.2 Binning and utility routines

- `int SPReadBinningFile(char *filename, struct BinFactors *binning)`

  Read an ascii file with binning factors and load the binning array.

- `int SPSetGroupArray(int inputSize, struct BinFactors *binning, int *groupArray)`

  Set up a grouping array using the BinFactors structure.

- `int SPBinArray(int inputSize, float *input, int *groupArray, int mode, float *output)`

  Bin an array using the information in the grouping array.

- `void SPsetCCfitsVerbose(int mode)`

  Set the CCfits verbose mode.

- `int SPcopyExtensions(char *infile, char *outfile)`

  Copy all HDUs which are not manipulated by this library.

- `int SPcopyKeywords(char *infile, char *outfile, char *hduname, int hdunumber)`

  Copy all non-critical keywords for the hdunumber instance of the extension hduname.

Appendix A

Ftools and Heasp

The following table lists ftools that operate on spectra or responses and the related HEASP routines. The read and write routines apply in all cases so are not included in the table. In some, relatively simple, cases the ftool equivalent could be performed by directly getting and setting class members.

Ftool	corresponding HEASP C++ routines
addarf	arf::operator+=, arf::operator+
addrmf	rmf::operator+=, rmf::operator+
cmppha	phaII::get
cmprmf	rmf::compress
dmprmf	directly access rmf class members
gcorpha	*none yet*
gcorrmf	*none yet*
marfrmf	rmf::operator*=, rmf::operator*
rbnrmf	grouping::load, rmf::rebinChannels, rmf::rebinEnergies
arf2arf1	arfII:get
ascii2pha	directly set pha class members
chkarf	arf::check, arfII::check
chkpha	pha::check, phaII::check
chkrmf	rmf::check
flx2xsp	directly set pha and rmf class members
grppha	grouping::load, pha::setGrouping
grppha2	phaII::get, grouping::load pha::setGrouping, phaII::push
mathpha	*none yet*
rbnpha	grouping::load, pha::setGrouping, pha::rebinChannels
rsp2rmfarf	directly get and set rmf and arf class members
sprbnarf	grouping::load, GroupBin

Appendix B

Error Codes

- 1 : NoSuchFile : Cannot find the file specified.
- 2 : NoData : A column read has no members.
- 3 : NoChannelData : The SPECTRUM has no Channel column and no channel data can be constructed.
- 4 : NoStatError : The SPECTRUM has no statistical error column and POISSERR=F.
- 5 : CannotCreate : Cannot create a new file.
- 6 : NoEnergLo : The ENERG_LO column in an ARF or RMF file has no data.
- 7 : NoEnergHi : The ENERG_HI column in an ARF or RMF file has no data.
- 8 : NoSpecresp : The SPECRESP column in an ARF has no data.
- 9 : NoEboundsExt : There is no EBOUNDS extension in an RMF file.
- 10 : NoEmin : The E_MIN column in an RMF file has no data.
- 11 : NoEmax : The E_MAX column in an RMF file has no data.
- 12 : NoMatrixExt : There is no MATRIX extension in an RMF file.
- 13 : NoNgrp : The N_GRP column in an RMF file has no data.
- 14 : NoFchan : The F_CHAN column in an RMF file has no data.
- 15 : NoNchan : The N_CHAN column in an RMF file has no data.
- 16 : NoMatrix : The MATRIX column in an RMF file has no data.
- 17 : CannotCreateMatrixExt : The output MATRIX extension cannot be created.
- 18 : CannotCreateEboundsExt : The output EBOUNDS extension cannot be created.
- 19 : InconsistentGrouping : The grouping information size is different from that of the array to which it is being applied.