DEPARTMENT OF SPACE RESEARCH
BIRMINGHAM UNIVERSITY
Asterix Data Analysis
ASTERIX Programming Note 015

PROG/015

David J Allan
23 Jul 1994

The Abstract Data Interface (ADI)

Contents

1 Introduction

Asterix applications currently perform their data interface using a collection of routines in ASTLIB (BDA,LIST,SSO,PSF,HIST to name the major ones) and low-level access calls in the native file representation (HDS). The number of instances of the latter is decreasing asymptotically as extra abstract routines are introduced. The major area in which low-level access is still performed is in the the access of pointing information and other header components.

Shortcomings in the interface routines are now becoming apparent on a number of fronts.

- FITS. Support for at least a well defined subset of the plethora of FITS formats will be essential if Asterix is to survive in the medium term. FITS is the only sensible machine-independent format for storing event data.
- NDF support. We gradually extend the BDA system to cope with the odd dataset from KAPPA etc. which uses aspects of the NDF definition we'd previously ignored. This has been done in a rather ad hoc fashion with the result (for example) that DATA_ARRAY.DATA is supported, but not VARIANCE.DATA.
- Simple programming interface. The commonly used interface routines in Asterix number between 100 and 150 which is far too many for a part-time programmer to be aquainted with, and even leads to professionals rewriting code because they are not aware that a particular function exists. Even if we could gather together all our interface routines and write a big document, we'd hardly dare call it a homogenous set with a common design philosophy running throughout.

2 The ADI Design Goals

The following summarises the ADI design discussed below.

- The number of ADI root routines (ie. ignoring those which differ only in the type of key arguments) should be kept as small as possible.
- The interface should allow advantage to be taken of particular gains in efficiency in accessing data in particular ways, eg. sequentially in FITS, or chunking of HDS arrays.
- The system should have as few static limits as possible (eg. on the number of datasets). By doing everything dynamically upper limits are removed, and the saving in memory in the far more usual case of operation near the lower limit is well worthwhile.
- The system should allow inheritance in the sense that if, for example, a new feature is added to the processing of HDS arrays, then that feature would be available to manipulate the data, variance and other array components of a binned dataset.

3 Users View of ADI

This section presents ADI from the view of the user, namely applications programmers *and* amateur programmers. The various concepts of the system are explained in an abstract form, the nitty gritty of the implementation is discussed in the next section.

Asterix analysis applications communicate with data residing in the file system using (mostly) subroutine interfaces which provide data abstraction, ie. the information regarding the format used in the underlying file is coded in the access layer rather than the application. This is only good computing practice – applications are insulated from changes in file formats. When a system evolves using only one data format it is not surprising that the elements of the interface layer map very closely onto components of the underlying data format, and so it is easy to confuse the data model with the data representation.

When new data formats are added one has to ask whether there is a fundamental change required in your data model. The answer for the OGIP FITS formats is clearly not. The requirements driving the definition of the OGIP FITS formats were largely the same as those driving the Asterix extension to the Starlink NDF format, and the differences largely reflect the flat vs. hierarchical file formats preferred in the two cases.

The task of constructing a unified abstract data interface is then one of defining an appropriate data model, and the mappings between the various file formats and that model.

3.1 The Class

Applications generally deal with data in a particular data model. The more general purpose the program the fewer requirements it makes that all elements of the data model be present. A subsetting program doesn't need to access astrometric data for example, but a source searching program certainly does.

ADI provides a environment for defining data models, the operations on those models and interfaces with the file system (if any). A particular data model is known as a *class*. [1] Applications using ADI manipulate *instances* of a class, which correspond to particular datasets or other data objects. The creation of a new instance of a class is often called *instantiation*. For an application to be usable across multiple file formats it should access *only* the abstract data model, or at least make the requirement for data outside the model optional.

The class most frequently accessed by Asterix applications will be the binned dataset, which has a *class name* of `BinDS`. A class is a container for other named data objects called *data members*, each of which can itself be the instance of a class, or simply primitive data (integers, reals and so on). The crucial point is that data members need be neither present, nor have counterparts in the file format if they are. Such data members will often be referred to as *virtual* data members. Generally applications are unconcerned whether data are "real" or "virtual". A good example of a virtual data member is the `Error` array of a `BinDS`. In the NDF format their exists a variance array which ADI presents as a virtual data member by taking the square root of the values.

The ADI class system also exhibits *inheritance*, meaning that new classes can be derived from existing classes. Inheritance can use be used in two subtly different ways,

[1] The similarity with OOP nomenclature should be considered a hint for those familiar with such things.

- Extending the data model
- Constraining behaviour

In the first new components are added to those of existing class. The new class inherits all the data from the its parent or *superclass*. [2] This is not used that much in the proposed ADI design (and is only present because the method of implementation is identical to the second). Constraining behaviour allows a new class to be defined where the the values of the class data members can take on only a subset of the values in the superclass. This subsetting can occur explicitly be setting the values of data members to specific values in the new class definition, or implicitly be allowing procedures which operate on the new class to constrain the values of those members. The example below demonstrates both aspects of inheritance using a pseudo-class definition,

```
CLASS Array                         ; top-level class, no inheritance
  _INTEGER    ndim
  _INTEGER    dims[ndim]
  <NUMERIC>   data[...]
END CLASS

CLASS MagicArray INHERITS Array     ; Extend the Array data model
  <NUMERIC>   magic                 ; new member 'magic' to hold array
END CLASS                           ; special value

CLASS VectorisedArray INHERITS Array ; Constrain the existing class
  _INTEGER    ndim = 1
END CLASS
```

The `MagicArray` shows the first form of inheritance, the `VectorisedArray` the second. The latter form is heavily used in ADI to provide useful constraining classes for applications. The constraints built in to the derived class can relieve much of the burden of checking inputs from applications.

3.2 Class Methods

Class data members have been discussed above in some detail, but nothing has yet been said about how they are used. ADI allows procedures or *methods* to be defined which can be overloaded on the basis of the class of their arguments, ie. two methods can be defined with the same name which have different arguments. [3] ADI reserves some method names for its own use for particular operations, and by specifying the Fortran subroutines or C functions that will be called when these methods are required, functionality is added to the ADI system. Not much more can be said about methods for the moment until their place in the system becomes apparent in the sections that follow.

[2] A class can actually have multiple superclasses.
[3] This feature is known as *polymorphism*.

3.3 Classes and Files

A number of different topologies of ADI objects with respect to the application are possible. Figure 1 shows the three varieties. In the first of three schemes the ADI class instance has no

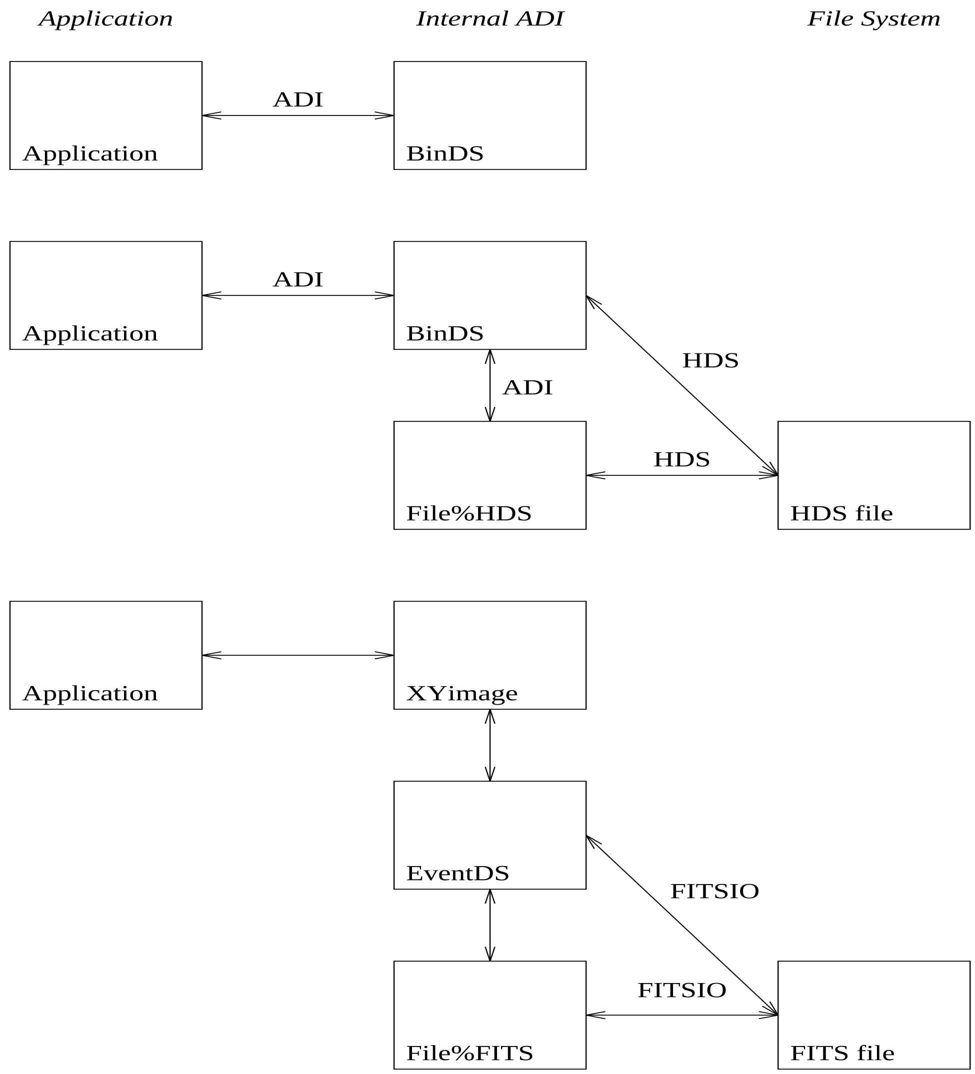

Figure 1: Example ADI object topologies

connection to the file system. In this case the the object resides purely in memory, ie. it is a virtual dataset. Asterix currently supports dynamic numeric arrays, but this facility allows complete dynamic datasets to be constructed. It also allows the possibility of testing class behaviour before the the file interface is written.

The second example shows the most common case where an ADI object is mediating exchanges between the application and the underlying file system. When a file interface is required ADI creates an instance of a class `File` which stores the file format specific information needed to control the file. The `BinDS` object communicates both with the `File` and with the file format directly using class methods. The trailing `%HDS` on the `File` box will be explained soon.

The third example shows a more complex setup where an intermediate ADI instance of `XYimage` has been interposed between the application and the `EventDS` object closest to the `File`.The ADI system allows such chains to be constructed and controls the communication between such objects. The application in this example could handle inputs of event datasets when it was expecting an image as long the criteria for setting up the network of objects had been met (which would boil down to spatial fields existing in the event list).

3.4 File Representation

ADI has a concept of *file representation*. Different representations correspond to fundamentally different file formats, such as HDS, FITS or ascii text. When you specify a filename chances are there will be only one valid interpretation of that filename, but this is not guaranteed. For example, you could have an ascii file called `myfile.data` and an HDS file called `myfile.sdf` with a top-level component called `data` in the same directory. The behaviour of ADI when a filename ambiguity occurs is discussed in Section **??**. The mechanism for resolving the ambiguity is to append the file representation code to the filename, for example,

```
myfile.data%ascii
myfile.data%hds
```

The percent character is the file representation delimiter. The syntax for specifying HDS objects is the usual ADAM one, some examples of which are,

```
myfile                        ; myfile.sdf top level, extension assumed
myfile.data_array             ; a top-level component
myfile.variance(1:10,2:*)     ; a slice of an array
```

The syntax for FITS is the same as that used by FTOOLS,

```
myfile                        ; the whole FITS file - .fits is assumed
myfile.pha[1]                 ; whole name of file needed, use first
                                FITS extension only
myfile.pha+1                  ; Alternative version of above
myfile.pha[1]%fits            ; as above if ambiguous representation
                                (not very likely!)
myfile.pha^BITPIX             ; specific keyword
myfile.pha[3]^NEVENTS         ; specific keyword in extension
```

The default file extension may differ from `.fits` depending on the ADI class requested. For example, XSPEC format spectra could have a default file extension of `.pha` defined for a class `Spectrum`.

Once ADI has opened the named file object it attempts to derive its *natural class*. This is implemented as sequence of rules mapping attributes of the opened file (such as HDS type or the FITS HDU class keywords) to ADI class names. If a natural class can be defined, and that class is the one requested by the application then the search is over. If the natural class is different, then ADI searches for intermediate routes to go from the natural class to the requested class, constructing an analogue of the topology shown in the third example on Figure 1. In the absence of a natural class ADI attempts to connect the file object directly to an instance of the class requested.

File representation can be propagated in different ways, depending on the way in which output files are derived. Options can range from "all outputs must be HDS" type statements to "inherit the representation from the input file if the output is derived from it, otherwise use FITS". This can all be done in a fashion which is class dependant (you might always want spectra in XSPEC format, but images in HDS for example).

3.5 Primitive Data, Classes & Aggregates

ADI data objects can be divided into three groups. The first group are the primitive data objects. All the primitive HDS types are supported and have the same names (`_UBYTE`, `_REAL` etc), but character strings are of variable length and are known simply by the name `_CHAR`. Primitive objects are *not* instances of a class. This means that you cannot place the name of a primitive type in a class inheritance list.

The second data group are instances of ADI classes. A class instance is a collection of data objects of three forms,

- Class member data. Can be split into two catagories which exist in the same name space.
  - Mandatory keywords. These are the quantities which are required before an instance of a class can be created. For example, the keywords for the class `Array` are SHAPE and TYPE. The first is a 1-D integer array and the second a character string.
  - Useful data members, ie. the information an application is interested in accessing. Each class member has a name which exists in the same name space sa the keywords above.
- Properties. Every class instance has a property list. This is simply a list of (name,value) pairs which can be indexed by the name. The property list is used for stuffing useful bits of information by ADI. User software can also use the property list.
- ADI header data. This is just the information ADI stores to keep track of each object in the system. The main things are an RTTI (Run Time Type Identifier) which tells ADI which class the data belongs to, and the binding field which controls the link from an instance to other ADI class instances (such as file objects).

Class data members and property values can be either primitive data or class instances. There is no limit to the numbers of any of these data members, other than those imposed by the process quotas. The ADI header data is not really for use by applications and no routines are provided for its direct manipulation.

The third data group are the aggregates. The most useful aggregate is the array (not to be confused with the `Array` class). Arrays may consist of any primitive type or class instances. Two other aggregates are lists and hash tables. The system is flexible enough to allow the creation of third party data aggregates (eg. bags or sets) and the specification of the rules for creating and disposing of their contents.

3.6 Data Access

Historically Asterix applications have used only one data access method for bulk data handling, namely file mapping. This has three disadvantages,

- It encourages the mode of programming where all the data is read in, manipulated, and then perhaps written back out. While this method is sometimes required, it is very inefficient for large datasets. For many applications a single pass with a modest buffer would suffice.
- Mapping returns a pointer to a place in the process' address space as an integer which must be dereferenced using the `%VAL` function before the data can be used at a low level. This inevitably forces programs to be split into two levels. The combination of the mapping process and the resulting requirement to split the code makes programming noddy analysis procedures unnecessarily complicated.

For these reasons ADI supports two data access modes in addition to mapping. The first of these is a simple `get/put` interface where the entire data object is read into or written from Fortran variables in one fell swoop. This makes the life of the amateur programmer considerably simpler.

The third access mode is buffered access. ADI buffers can be defined to scan over a data object or objects with buffer sizes under either user or automatic control. Access to buffered objects can be synchronised to (for example) step over multiple data members of a class at the same rate.

4 An Example Class

As an example of the ADI system we shall define and construct a few useful routines for a class called "Array". The definition first,

```
data      members/'SHAPE*','TYPE*','Data'/

adi_defcls( 'Array', ' ', 3, members, status )
```

The first argument to the routine `adi_defcls` is the name of the new class, followed by the names of the classes whose behaviour the new class will inherit (in this case none). The number and definition of the class data members follows. The asterisk following the names of the first two members indicates that they are mandatory keywords. If a data member has a default then this may be inserted after the keyword name following an equals sign, eg.

```
data      members/'SHAPE*','TYPE*=_REAL','Data'/

adi_defcls( 'Array(SHAPE,TYPE=_REAL)', 1, 'Data', status )
```

Note that ADI primitive types have the same names as their HDS counterparts. The rest of this discussion will assume that we have used the first of the definitions above.

The instantiation of all but the most trivial class objects is a two stage process. This is because the information required to perform create a functional instance of a class is arbitrarily large. The first stage is to allocate an empty template for the class.

```
adi_new( 'Array', aid, status )
```

The only purpose of this routine is to reserve ADI internal storage space and to supply the user with an ADI identifier `aid` which can be used to further manipulate the object. The next stage is to define the values of the mandatory keywords,

```
adi_cput1i( aid, 'SHAPE', 1, 256, status )
adi_cput0c( aid, 'TYPE', '_REAL', status )
```

All the information required to create an array has now been supplied. All that remains is to create and perform the binding to any external objects as required. To perform a null binding (ie. the array will reside in memory) just specify `0` as the second argument to `adi_creat`.

```
adi_creat( aid, 0, status )
```

We now have a new class instance up and running which we can shove data into. Here are a few examples,

```
adi_cput1d( aid, 'Data', 256, my_double_array, status )
adi_get1i( aid, 256, ivals, status )
```

The first of these demonstrates that ADI performs on the fly type conversion just like HDS. It is rather annoying to have so specify 'Data' when we know that there's only one data member, so ADI lets you use the primitive routines in this circumstance – this is the only occasion where these routines can be thus used.

4.1 Connecting to Files

The binding mechanism enables ADI class instances to map onto file system objects such as HDS components, or FITS arrays, extensions or keywords. The easiest way to understand is by way of an example. The `Array` class will be extended to handle HDS primitive arrays and FITS primary arrays.

The routine ADI uses to connect existing file system objects to an ADI class is `adi_fopen`, which looks like this,

```
adi_fopen( fspec, mode, class, id, status )
```

where `fspec` is the path to the file object, `mode` is the access mode and `class` is the ADI class we expect to open. For our `Array` example we want something like,

```
adi_fopen( 'myfile.data_array', 'READ', 'Array', aid, status )
```

ADI opens the object named in the first argument with access mode provided in the second, but is then stuck. It has to know how to connect the fact that it has a locator to a primitive HDS array to an ADI class instance. ADI has a concept of a "natural class" for a file object, ie. that ADI class which most closely matches the way the data is stored on disk. We can extend ADI's knowledge of these matches at run-time by defining a *callback* routine. A callback is simply a routine which is not executed immediately, but later when it is needed. The way this is done is,

```
external adi1_isarray

adi_defrcb( 'HDS', 'Natural', adi1_isarray, status )
```

The `adi_defrcb` routine defines a callback for a particular file representation. The callback function *must* have one of the predefined names listed in Appendix **??**. The callback routine has an extra digit after the `adi` and before the underscore. This flags it as an internal routine, and the value indicates that it is HDS specific. Callback routines have fixed argument lists and the definition for a `Natural` callback is the following,

```
<natural_callback>( id, class, status )
  integer       id
  character*(*) class
```

The `id` argument is the identifier of the `File` ADI has connected to your named file. The purpose of the routine is to try and derive a value for `class`. Here is a pseudo-code version of `adi1_isarray`,

```
subroutine adi1_isarray( fid, class, status )
  integer       fid
  character*(*) class

  extract locator from FID
  use HDS routines DAT_PRIM and DAT_SHAPE to see if the object is
    a primitive array
  if it is,
    class = 'Array'
  else
    class = ' '
end
```

ADI now knows that the user has requested an `Array` and that it has a file object whose natural class is also `Array`Ṗutting 2 and 2 togther ADI now has to create the instance of an `Array` class which has a binding to the `File`. This is done using another callback, but one which is specific to a particular class.

```
external ari1_open

adi_defccb( 'Array%HDS', 'Open', ari1_open, status )
```

The purpose of this routine is to write the values of the mandatory keywords into the empty `Array` instance. The syntax of the first argument tells ADI that this routine should only be applied to an `Array` object bound to an HDS based `File`.

```
subroutine ari1_open( fid, id, status )
  integer fid,id

  extract locator from FID
  use DAT_SHAPE and DAT_TYPE to define the SHAPE and TYPE keywords
end
```

There are two fundamntal ways of implementing file interface objects. The first and by far the simplest to program is imply to copy all the file data objects into memory on read, writing them out again if required. In reality this would be far too slow and use too much memory and a complex scheme which converts ADI accesses to the class instance object into file system access calls is required.

To make the mapping between the memory based `Array` object and the file based HDS array useful we must now overload the the various data access methods used by ADI. To simply read the values from a class data member the `adi_mget` routines are used, eg.

```
adi_mget1r( aid, 'Data', maxval, my_reals, nactval, status )
```

The `adi_mget` routine is implemented by invoking the `Get` method on the named class data member. This can be overloaded using the callback mechanism,

```
adi_defccb( 'Array%HDS', 'Get(Data)', ari1_getdata, status )

subroutine ari1_getdata( id, mem_hdl, status )
  integer    id
  integer    mem_hdl

  get locator from ID
  check data is valid
  get memory space TYPE, DIMS and PTR from MEMHDL
  DAT_GET the data to the address PTR
end
```

5 Implementation Notes

5.1 C and Fortran interfaces

The ADI kernel is written in ANSI standard C with access to Fortran through Starlink's F77 package.

Both C and Fortran programmers interfaces are provided. The Fortran routines are prefixed with `ADI_` and the C routines with `adic_` (in lower case). The routine names are otherwise the same, although the arguments may differ slightly, especially where character strings are concerned.

5.2 Representation Subclassing

Representation can be subclassed by appending `.<subrep>` to the representation name, eg. `fits.eso`. This means that ADI uses the file open and closure routines for representation `FITS`, but class methods can be overloaded on subclassed representation to allow for differences in, say, FITS header keywords between MIDAS and FTOOLS.

5.3 ADI Configuration

6 The ADI Classes

The following sections list the classes currently in the ADI design and the mappings between some of the commonly used file formats in the class data members. In the tables mandatory keywords are printed in upper case in the upper half of the table (if present), and class data members are shown in the second half. Data members which are mandatory are shown in **bold** typeface.

6.1 The `Scalar` Class

The class `Scalar` is the simplest class [4], consisting of only one data member which has a name `Value`.

Member Name	**Type**	**Description**
TYPE	_CHAR	ADI type for new Scalar value
Value	TYPE	The single data value

[4] Not quite true, see Section XX later

6.1.1 HDS mapping to `Scalar`

The mapping from HDS to the `Scalar` class is very simple – all primitive scalar HDS components can be used as objects of class `Scalar`.

6.1.2 FITS mapping to `Scalar`

FITS keywords can be used wherever a `Scalar` value is used.

6.2 The `Array` Class

The `Array` class is the most important for Asterix data analysis. The controlling attributes of an `Array` are its dimensions and type. Arrays may have arbitrary dimensionality, although limitations are imposed by both HDS and FITS in representing arrays of high dimensionality in their respective file formats.

Member Name	**Type**	**Description**
SHAPE	_INTEGER[]	Dimensions of array
TYPE	_CHAR	ADI type for new Scalar value
Data	TYPE[SHAPE]	The single data value
Origin	_INTEGER[]	Origin of first pixel per dimension
SpacedData	TYPE[2]	The base and offset of the spaced array

6.2.1 HDS mapping to `Array`

The HDS interface to `Array` maps primitive HDS arrays and structured objects of type `ARRAY` to ADI objects. Until such time as the Starlink ARY package supports the NDF fully it is envisaged that only arrays of variant `SIMPLE`, `SPACED` and `SCALED` will be supported.

6.2.2 FITS mapping to `Array`

FITS data units where `CONTENT` has the value `"IMAGE" are recognised as primitive arrays. The size of the $i^{th}$ dimension is given by the value of keyword \verb`NAXIS+i. Note that proper images will generally have a natural class of `BinDS` or `XYimage` due to the presence of axis values and other ancillary information.

6.2.3 The `VectorisedArray` Subclass

The `VectorisedArray` is a subclass of `Array`.

6.3 The `BinDS` Class

A binned dataset is a collection of `Array` objects in which dimensionality information is shared across certain arrays. Arrays attached as properties to BinDS objects can have the property "BinDSconformal" which means that these arrays are subject to shape change under subsetting and rebinning operations. In these circumstances it is possible that normalisation properties are important. In general this will not be the case however, as this facility will be used to store plot attributes on a per point basis.

Member Name	**Type**	**Description**
SHAPE	_INTEGER[]	Dimensions of array
TYPE	_CHAR	ADI type for new Scalar value
		Has a default of `_REAL`.
Data	`Array` of TYPE	The primary data array
Axes	struct []	Array of axis structures
Data	`Array` of TYPE	Axis data values
Width	`Array` of TYPE	Axis bin widths
Normalised	_LOGICAL	Data values normalised wrt this axis?
Label	_CHAR	Axis label
Units	_CHAR	Axis units
Variance	`Array` of TYPE	The data variance values
Error	`Array` of TYPE	The data error values
SystematicError	`Array` of TYPE	The systematic error values
StatisticalError	`Array` of TYPE	The statistical error values
Quality	`Array` of _UBYTE	Data quality values
QualityMask	_UBYTE	Data quality mask
History	`Array` of _CHAR[]	History records
Title	_CHAR	Dataset title string
Label	_CHAR	String describing contents of `Data`
Units	_CHAR	String describing units of `Data`

In the case where both `SystematicError` and `StatisticalError` are defined, the `Error` component will return the two errors added in quadrature, and `Variance` the square of that quantity. [5]

6.3.1 HDS mapping to `BinDS`

The HDS interface to the `BinDS` class is well defined apart for those data members which are required to support some features of the OGIP format for spectral data. In that format systematic and statistical errors can be handled separately. The solution adopted will be that for HDS the `SystematicError` component would not be present, and the `StatisticalError` component will give the same data as the `Error` component.

[5] The Asterix application SYSERR will be updated to handle `SystematicError` if present

6.3.2 FITS mapping to `BinDS`

6.3.3 Ascii mapping to `BinDS`

The Asterix file format used by IMPORT and EXPORT is a useful way of getting all the important attributes of a HDS binned dataset into ascii, where it can be e-mailed, edited or otherwise manipulated. By simply moving the code for IMPORT and EXPORT into ADI, actor objects will give direct access to ascii files by applications using ADI without the requirement to perform the format conversion (although it would be recommended for large datasets). Because the ascii file used is read and written sequentially it would be quite possible for Asterix applications to use UNIX pipes to pass data around. The syntax adopted for this would follow UNIX idiom, ie. the hyphen means use standard input for read operations and standard output for write operations. An example where you want to find the histogram of a dataset and display it to a device without keeping the intermediate file would be,

```
% frequency my_cube - reg spacing=2 | gdraw - xwindows
```

where the first hypens sends the output of the `FREQUENCY` command to standard output, and the second tells `GDRAW` to read from standard input.

6.3.4 The `VectorisedBinDS` Actor

6.3.5 Subclasses of `BinDS`

```
Spectrum            - Single spectrum
SpectrumSet         - multiple arrays, indexed
SpectrumArray       - one big array, 1st dimension spectral
Image               - X,Y image
ImageArray          - X,Y,... array
```

6.4 The `EventDS` Class

Member Name	**Type**	**Description**
Organisation	_CHAR	Storage mode. Can take values "FieldOriented" or "RecordOriented" so far
NEVENT	_INTEGER	Number of events in dataset
Fields	struct []	Array of structures holding field info
Data	`Array`	Field data values
LegalMin/Max	`Scalar`	Legal minima and maxima of field value
DataMin/Max	`Scalar`	Actual minima and maxima of field values
Quantum	`Scalar`	Smallest unit of field
Label	_CHAR	Label identifying field contents
Units	_CHAR	Physical units of field
Records	struct []	Event records in array form
Title	_CHAR	Dataset title string
History	`Array` of _CHAR[]	History records

6.5 Header & Ancillary Information

Member Name	**Type**	**Description**
Observation		Observation book-keeping details
Telescope	_CHAR	Name of whole observatory
Instrument	_CHAR	Name of instrument on observatory
Detector	_CHAR	Name of detector on instrument
Filter	_CHAR	Name of filter on detector
Observer	_CHAR	Name of observer
Object	_CHAR	Object name of observation
CatalogueID	_CHAR[]	Alternative names
Pointing		
System	_CHAR	Stellar reference frame, should be `FK4` or `FK5`
Equinox	_REAL	The date in decimal years of the Besselian epoch or Julian epoch appropriate for the specified `System`
ObjectRA/DEC	_DOUBLE	Position of the source of interest
PointRA/DEC/PA	_DOUBLE	Pointing direction and PA of optic axis
Timing		Observation times & exposure
ObsStartDate	_CHAR	Date string of form yyyy-mm-dd
ObsStartMJD	_INTEGER	MJD of observation start
ObsStartUTC	_DOUBLE	Fractional MJD of observation start
ObsEndDate	_CHAR	Date string of form yyyy-mm-dd
ObsEndMJD	_INTEGER	MJD of observation end
ObsEndUTC	_DOUBLE	Fractional MJD of observation end
ExposureTime	_DOUBLE	Exposure time in seconds
Asterix		Data specific to Asterix software
GCB	_CHAR[]	Graphics control block

6.5.1 HDS and OGIP FITS header info

The following mapping exists between Asterix HDS header values, OGIP FITS defined keywords and ADI class objects, class values,

Asterix NDF	**OGIP Keyword**	**ADI component**
HEADER.OBSERVATORY	TELESCOP	Observation.Observatory
HEADER.INSTRUMENT	INSTRUME	Observation.Instrument
INSTRUMENT.DETECTOR	*none*	Observation.Detector
HEADER.FILTER	FILTER	Observation.Filter
HEADER.TARGET	OBJECT	Observation.Object
assumed from below	RADECSYS	Pointing.System
HEADER.EQUINOX	EQUINOX	Pointing.Equinox
HEADER.FIELD_RA	RA_OBJ	Pointing.ObjectRA
HEADER.FIELD_DEC	DEC_OBJ	Pointing.ObjectDEC
HEADER.AXIS_RA	RA_PNT	Pointing.PointRA
	RA-NOM	
HEADER.AXIS_DEC	DEC_PNT	Pointing.PointDEC
	DEC-NOM	
HEADER.POSITION_ANGLE	PA_PNT	Pointing.PointPA
HEADER.ROLL	DROLLANG	Pointing.PointPA (converted)
HEADER.BASE_DATE	DATE-OBS	Timing.ObsStartDate
HEADER.BASE_MJD	DATE-OBS	Timing.ObsStartMJD
HEADER.BASE_UTC	TIME-OBS	Timing.ObsStartTime
HEADER.BASE_DATE* + OBS_LENGTH	DATE-END	Timing.ObsEndDate
HEADER.BASE_MJD* + OBS_LENGTH	DATE-END	Timing.ObsEndMJD
HEADER.BASE_MJD* + OBS_LENGTH	TIME-END	Timing.ObsEndTime
HEADER.EXPOSURE_TIME	EXP_TIME	Timing.ExposureTime
	EXPOSURE	

6.5.2 Astrometry

Astrometric information can be split in two. For those data where there is no spatial component a position on the sky is all that is required. Data which has spatial axes may have a mapping between either pixel number or axis units and a celestial coordinate frame.

Astrometric information is handled in ADI by the class `Pointing`. It has the following definition,

```
CHAR      SYSTEM        FK4, FK5
DOUBLE    EQUINOX
DOUBLE    EPOCH
DOUBLE    RA,DEC,ROLL
```

6.6 Asterix NDF

The only existing astrometry definition in Asterix NDF's is a gnomic projection where the celestial origin is defined by the header parameters AXIS_RA, AXIS_DEC and POSITION_ANGLE.

7 Conclusion

That portion of ADI which deals with class definition and the allocation of data objects is complete. A demonstration of the system is being constructed which will provide an Asterix – XSPEC two-way conversion by storing spectra and respones within an ADI class framework. This should be done in about TBD weeks.

8 References

1. Currie, M.J., Wallace, P.T., Warren-Smith, R.F., Starlink Standard Data Structures. bf SGP/38.2. January 1989.

A Classified Routine List

Routines have been broken up into groups according to function.

A.1 Extending the Class System

A.1.1 ADI_DEFCLS

```
Interface

  adi_defcls( classname, superclasses, nmember, members, status )

    >character*(*)     classname             ; the new class name
    >character*(*)     superclasses          ; superclass list
    >integer           nmember               ; number of data members
    >character*(*)     members(*)            ; description of data member

Description

  Defines the named class. Superclass names separated by commas, or
  a blank string if none. Number of data members can be zero. Data
  member strings are of the form

         [<type|class>] [superclass::]<name>[*] [= <value>]

  The optional type or class name constrains the type of the data
  member or class. The asterisk indicates a mandatory keyword which
  if not defined will result in \verb+Create+ methods failing. The
  optional value supplied is a default starting value for the member.

Errors
```

```
ADI__INVDEF          Class name already defined
                     Superclass named does not exist
                     Invalid class name, data member name
```

A.2 Creating and Destroying Objects

A.2.1 ADI_NEW

```
Interface

  adi_new( type, id, status )

   >character*(*)      type                   ; the type or class name
   <integer            id                     ; the new object identifier

Description

  Allocate space for a single object of the specified class or type.
  This routine can create a new primitive or a new class instance.

Errors

  ADI__INVTYP          Unknown ADI type or class name
```

A.2.2 ADI_NEW1

```
Interface

  adi_new1( type, n, id, status )
  adi_new1<t>( n, id, status )

   >character*(*)      type                   ; the type or class name
   >integer            n                      ; number of elements
   <integer            id                     ; the new object identifier

Description

  Allocate space for a 1-D array of objects of the specified class or
  type. This routine can create a new primitive or a new class instance
  array. The routines with the standard single letter HDS type codes
  appended create primitives of the indicated type.

Errors

  ADI__INVTYP          Unknown ADI type or class name
```

A.2.3 ADI_NEWN

Interface

```
adi_newn( type, ndim, dims, id, status )
adi_newn<t>( ndim, dims, id, status )

 >character*(*)     type                ; the type or class name
 >integer           ndim                ; number of dimensions
 >integer           dims(*)             ; number of elements
 <integer           id                  ; the new object identifier
```

Description

Allocate space for a n-D array of objects of the specified class or type. This routine can create a new primitive or a new class instance array. The routines with the standard single letter HDS type codes appended create primitives of the indicated type.

Errors

```
ADI__INVTYP         Unknown ADI type or class name
```

A.2.4 ADI_ERASE

Interface

```
adi_erase( id, status )

 >integer           id                  ; the object identifier
```

Description

Destroy an object and return memory resources allocated to it. If the object is a class instance then all the class data members are destroyed and any destructor methods invoked.

Errors

```
ADI__INVOBJ         Invalid ADI object identifier
```

A.3 Data Access

A.3.1 ADI_GET

Interface

```
adi_get0<t>( id, value, status )

 >integer           id                    ; the object identifier
 <type              value                 ; object value
```

Description

Returns the value of a primitive scalar object, or of a class instance with only one data member whose value is scalar.

Errors

```
ADI__INVOBJ          Invalid ADI object identifier
```

A.3.2 ADI_GET1

Interface

```
adi_get1<t>( id, maxval, value, status )

 >integer           id                    ; the object identifier
 >integer           maxval                ; max number of values to read
 <type              value(maxval)         ; object values
 <integer           actval                ; actual values read
```

Description

Returns the value of a 1-D object, or of a class instance with only one data member whose value is 1-D.

Errors

```
ADI__INVOBJ          Invalid ADI object identifier
```

A.3.3 ADI_GETN

Interface

```
adi_getn<t>( id, mxndim, maxdims, value, actdims, status )

 >integer           id                    ; the object identifier
 >integer           maxndim               ; max dimensionality to read
 >integer           maxdims(*)            ; max number of values to read
 <type              value(maxval)         ; object values
 <integer           actndim               ; max dimensionality to read
 <integer           actdims(*)            ; actual values read
```

```
Description

  Returns the value of a n-D object, or of a class instance with only
  one data member whose value is n-D.

Errors

  ADI__INVOBJ          Invalid ADI object identifier
```

A.3.4 ADI_PUT

```
Interface

  adi_put0<t>( id, value, status )

   >integer            id                    ; the object identifier
   >type               value                 ; object value

Description

  Sets the value of a primitive scalar object, or of a class instance
  with only one data member whose value is scalar.

Errors

  ADI__INVOBJ          Invalid ADI object identifier
```

A.3.5 ADI_PUT1

```
Interface

  adi_put1<t>( id, maxval, value, status )

   >integer            id                    ; the object identifier
   >integer            nval                  ; max number of values to read
   >type               value(nval)           ; object values

Description

  Sets the value of a 1-D object, or of a class instance with only
  one data member whose value is 1-D.

Errors

  ADI__INVOBJ          Invalid ADI object identifier
```