SCHOOL OF PHYSICS AND ASTRONOMY
UNIVERSITY OF BIRMINGHAM
DEV/001

Asterix Data Analysis
Development Note 001

Richard Beard
June 23, 1998

Replacing ADI within ASTERIX

Contents

1 Introduction

At the very heart of the ASTERIX system lies a set of library routines called the *Abstract Data Interface* (ADI). These routines were created by David Allen (DJA) to facilitate the storage of information and data independent of any external file format.

This is done through a set of pointers to either raw data types or structures of mixed data types. It also includes a scheme to allow for the calling of object oriented methods (ie. to implement procedures such as data output independent of the file type to be written to).

This document details why it might be necessary to replace the ADI library and outlines a scheme for doing so.

2 Why replace ADI?

When DJA left Birmingham University much of the knowledge on the workings of ADI went with him. The overall status of ADI at that time was one of only partial completeness, and thus the author spent many months of programming effort to obtain a working system.

This work was primarily in the outer layers of ADI, those that deal with the file dependent data I/O, and more specifically with the handling of Flexible Image Transport System (FITS) data files.

The part of ADI which handles the internal data storage and method handling was (thankfully) fully functional and required little alteration (apart from minor changes for a Linux port). This was fortuitous, as the large section of code with forms these routines is written in C, in a complex and largely undocumented manner. Whilst very competently coded indeed, it is felt that such a scheme is unsupportable within the ASTERIX system, and should for any reason this library cease to function correctly it would be difficult to maintain.

Thus, whilst everything functions correctly as the moment it is deemed necessary to anticipate any future problems by designing a new library to replace ADI.

3 Which parts of ADI to replace

There are three aspects of ADI that are deemed to be undesirable.

- *Object oriented methods* - almost all methods within ADI are to do with file I/O. They require a large amount of unsupported code that can simply be replaced with an `IF-THEN-ELSE` clause.
- *Internal data storage* - ADI uses its own private functions to facilitate the storage and retrieval of internal data. Starlink supplies such function in a number of packages which could be used.
- *Multiple types and checking* - The higher (file type dependent) ADI code performs many repeated checks on operations and allows for the limited case of multiple requests. The removal of both these features allows for much simpler code without seriously affecting the flexibility of ASTERIX.

The first two points are for the reasons outlined in the previous section, the last point is that a simplification of the system that can be performed at the same time for little extra effort in either planning or implementation.

The above scheme will also requires the replacement of all BDI calls, most particularly for points one and three. Thus every user application will have to be altered in the ASTERIX monoliths.

4 How to replace ADI

Some work has already been undertaken to replace ADI with calls to the *Abstract Data Functions* (ADF) library. See the directory `/lsoft1/asterix/work/rb/adf` for examples.

Stated simply, all calls to ADI have been replaced with identical calls to ADF. This allows for the minimum of change to the structure of applications and subroutines. The task is then to devise black-box replacements to perform the identical function as the old ADI call.

Also, all calls to BDI have been replaced with identical calls to BDF for the same reasons. The majority of this work is a simple find and replace of the string "ADI" with "ADF" and "BDI" with "BDF".

To date around 50packages that call these routines (grf, fit, hsi, &c) have not. Currently working versions of STATISTIX and BINLIST exist, along with a partial conversion of REBIN.

4.1 Object oriented methods

Currently an ASTERIX application will decided how to read some data from a file using a call such as (see file `bdi_get.f`),

```
CALL ADI_FEXEC( 'FileItemGet', 3, ARGS, DID, STATUS )
```

This will execute the subroutines `BDI1_GET` or `BDI2_GET` depending on whether the opened file is of type HDS or FITS. All the underlying code to define and implement these methods can be replaced with the following simple section of code,

```
CALL ADF_GETREP( ID, REP, STATUS )
IF ( REP .EQ. 'HDS' ) THEN
  CALL BDF1_GET( ID, LID, ITEM, DID, STATUS )
ELSE IF ( REP .EQ. 'FITS' ) THEN
  CALL BDF2_GET( ID, LID, ITEM, DID, STATUS )
END IF
```

N.B. Some more checking and sub-division of calls might be required if an operation depends on image, spectrum or time series data being present. It is expected that this can be accommodated by using an added complexity in the `IF-THEN-ELSE` clause.

4.2 Internal data storage

The ADI code creates its own sophisticated scheme for data storage, caching and memory management. It is this part of the library which will be the hardest to support, and thus the most important part to replace.

This proposal uses the Starlink HDS (Hierarchical Data Storage) library to mimic these functions of ADI, as HDS includes support for all three types of operation.

Stated simply, each ASTERIX application opens a temporary HDS container file at the start of execution. The ADF library uses this file to store internal objected during execution of the application. Upon completion the HDS file is unmapped and unlinked before closing. Unless large volumes of data are to be stored in this temporary file, it is expected that all HDS operations will be performed in memory, and thus add little in the way of processing overheads to any ASTERIX application.

NB. Currently the temporary file is created in `$PWD`, though it is expected this will be moved to `$ADAM_USER` in the future. Also, the appropriate code that performs the unmapping, unlinking and closing of both temporary storage items and the file itself has been commented out - to aid in development and de-bugging.

4.2.1 Opening the temporary file

Each application opens a temporary HDS container with the name `ADF_nnn.sdf`, where `nnn` is the process id of the application. This should provide enough scope to avoid name clashes, and if not, the name of the calling application can be added to further decrease this possibility.

The subroutine to open this file is called indirectly from `AST_INIT`, where the following line is added,

```
*    Initialize ADF_CMN
      CALL ADF_INIT()
```

The subroutine `ADF_INIT` does two things, first it calls `ADF_OPEN` and second it resets the internal object pointer. A trace of the initial tempory file is,

```
HTRACE Version 2.2-0
 ADF_OBJ          <ADF_OBJ>      {structure}
   ADF_LOC          <ADF_LOC>      {structure}
```

The file is used for storing ADF objects, these objects are indexed from ADF pointers (stored as integers) using the look-up table stored in the ADF locator structure. The HDS locators of these two structures are kept in common blocks as defined in the include file `ADF_CMN`.

4.2.2 Using the temporary file

The file will be used by the low-level ADF calls such as ADF_[C]NEW[n][v], ADF_[C]GET[n] and ADF_[C]PUT[n]. The first use is likely to be for opening a file, where the subtoutine `ADF_FOPEN` contains the line,

```
CALL ADF_NEW0( 'FILE', ID, STATUS )
```

Here a temporary structure `FILE` is to be created to store information on the opened file. The subroutine performs the following steps,

- Increment the variable `ADF_OBJ_PTR` to obtain a new pointer.
- Convert this number to a string for use as a component name.
- Create a new structure as instructed in the top level (`ADF_OBJ`) using the pointer value as its name.
- Find the HDS locator to this structure and add it as an entry to the look-up table (`ADF_LOC`) under the same name.

Data can be added to this `FILE` structure using code such as,

```
CALL ADF_CPUT0C( ID, 'REP', REP, STATUS )
CALL ADF_CPUT0C( ID, 'LOCATOR', FILE_LOC, STATUS )
```

After all information on the opened file has been stored the temporary file will look something like,

```
HTRACE Version 2.2-0
 ADF_OBJ          <ADF_OBJ>      {structure}
   ADF_LOC          <ADF_LOC>      {structure}
     2                <_CHAR*15>     " 5 h           "

   2                <FILE>         {structure}
     REP              <_CHAR*3>      "HDS"
     LOCATOR          <_CHAR*15>     " 5 0           "
     MODE             <_CHAR*4>      "READ"
     CLASS            <_CHAR*11>     "BinDS|Array"
     TRACE            <TRACE>        {structure}
       NLEV             <_INTEGER>     1
       PATH             <_CHAR*3>      "IMG"
       FILE             <_CHAR*35>     "/lsoft1/asterix/work/rb/adf/img.sdf"

     SHAPE[2]         <_INTEGER>     100,100
```

The development subroutine `ADF_PRINT` can be used display the contents of a pointer in a similar manner to the call `ADI_PRINT`.

4.3 Multiple types and checking

The ADI system allows for multiple opperations on different data objects with a single call, such as to copy the title, label and units of a dataset from one file to another with the single call,

```
CALL BDI_COPY( IFID, 'Title,Label,Units', OFID, ' ', STATUS )
```

To increase the simplicity of the replacement code this capability has been removed from all calls (copying, mapping, reading &c.). Thus the above code needs to be replaced by,

```
CALL BDF_COPY( IFID, 'Title', OFID, ' ', STATUS )
CALL BDF_COPY( IFID, 'Label', OFID, ' ', STATUS )
CALL BDF_COPY( IFID, 'Units', OFID, ' ', STATUS )
```

The BDI routines contain other checks and safeguards, such as ensuring a mapping mode is either read or write, and comparing the types of two objects. Because of the many-tiered nature of BDI (`BDI_AXMAPR` calls `BDI_AXMPAP` calls `BDI_MAP` &c.) these checks can often be repeated.

The new BDF routines assume (in general) that the original call was correct in all aspects and perform no checking of operation, type, &c. This again greatly simplifies the replacement code.

5 Opening FITS files

Since the code for `ADI1_OPEN` and `ADI2_OPEN` must be re-written an opportunity presents itself to make the syntax for accessing HDS and FITS files more flexible and compatible with other software (ie. FTOOLS).

Most, if not all work need to be done with FITS access, and it is proposed the new subroutine `ADF2_OPEN` be able to cope with all the following forms for opeing the file `data.fits`,

- data
- data.fits
- data%fits
- data.fits[0]
- data.variance

NB. Not all the code for these different forms of syntax is yet in place.